Part Number 800-1117-01
Revision: C of 7th January 1984
For: Sun System Release 1.1

System Internals Manual

for the

Sun Workstation

Sun Microsystems, Inc.,
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

Revision History

Revision	Date	Comments
A	1 August 1983	First Release of this manual.
B	1 November 1983	Second Release of this manual.
C	7 January 1984	Third Release of this manual. Added a section on the Sun Workstation's CPU PROM Monitor. Revisions to the Device Driver Manual. Added a section on Using ADB on the Kernel.

System Internals Manual

Table of Contents

This manual provides several papers on the internals of the Sun UNIX System:

Network Implementation on the Sun Workstation

Table of Contents

Networking Implementation Notes

The Sun Workstation runs a version of the UNIX† operating system which has strong support for network communications. This document describes the internals of the networking support subsystem. See the *System Interface Overview* in the Sun *System Interface Manual* for a description of the user interface to the networking facilities.

1. Introduction

This report describes the internal structure of the networking facilities of the Sun Workstation version of the UNIX operating system. These facilities are derived from the networking facilities added at U.C. Berkeley in the Berkeley 4.2 release of the system. The system facilities provide a uniform user interface to networking and a structure which may be used by system implementors to add new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *System Interface Overview* at the beginning of the Sun *System Interface Manual*. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

† UNIX is a trademark of Bell Laboratories.

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal Address Representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
        struct sockaddr {
              shortsa_family; /* data format identifier */
              char sa_data[14];     /* address */
        };
```

All addresses belong to one or more ***address families*** which define their format and interpretation. The *sa_family* field indicates which address family the address belongs to, the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats

5. Memory Management

A single mechanism is used for data storage: memory buffers, or *mbufs*. An mbuf is a structure of the form:

```
struct mbuf {
      struct     mbuf *m_next;          /* next buffer in chain */
      u_long     m_off;                 /* offset of data */
      short m_len;                 /* amount of data in this mbuf */
      short m_type;                /* mbuf type (accounting) */
      u_char     m_dat[MLEN];           /* data storage */
      struct     mbuf *m_act;           /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define     mtod(x,t)  ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);

The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);

The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);

The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);

After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define    dtom(x)    ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

6. Internal Layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

6.1. Socket Layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *System Interface Overview* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
     short so_type;          /* generic type */
     short so_options;       /* from socket call */
     short so_linger;        /* time to linger while closing */
     short so_state;         /* internal state flags */
     caddr_t    so_pcb;             /* protocol control block */
     struct     protosw *so_proto;  /* protocol handle */
     struct     socket *so_head;/* back pointer to accept socket */
     struct     socket *so_q0;      /* queue of partial connections */
     short so_q0len;         /* partials on so_q0 */
     struct     socket *so_q;       /* queue of incoming connections */
     short so_qlen;          /* number of connections on so_q */
     short so_qlimit;        /* max number queued connections */
     struct     sockbuf so_snd;     /* send queue */
     struct     sockbuf so_rcv;     /* receive queue */
     short so_timeo;         /* connection timeout */
     u_short    so_error;        /* error affecting connection */
     short so_oobmark;           /* chars to oob mark */
     short so_pgrp;          /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

6.1.1. Socket State

A socket's state is defined from the following:

```
#define SS_NOFDREF          0x001    /* no file table ref any more */
#define SS_ISCONNECTED      0x002    /* socket connected to a peer */
#define SS_ISCONNECTING     0x004    /* in process of connecting to peer */
#define SS_ISDISCONNECTING           0x008/* in process of disconnecting */
#define SS_CANTSENDMORE     0x010    /* can't send more data to peer */
#define SS_CANTRCVMORE      0x020    /* can't receive more data from peer */
#define SS_CONNAWAITING     0x040    /* connections awaiting acceptance */
#define SS_RCVATMARK        0x080    /* at mark on input */

#define SS_PRIV             0x100    /* privileged */
#define SS_NBIO             0x200    /* non-blocking ops */
#define SS_ASYNC            0x400    /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULDBLOCK (the service request may be partially fulfilled, e.g. a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

6.1.2. Socket Data Queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        short sb_cc;            /* actual chars in buffer */
        short sb_hiwat; /* max actual char count */
        short sb_mbcnt; /* chars of mbufs used */
        short sb_mbmax;         /* max chars of mbufs to use */
        short sb_lowat; /* low water mark */
        short sb_timeo; /* timeout */
        struct      mbuf *sb_mb;    /* the mbuf chain */
        struct      proc *sb_sel;   /* process selecting read/write */
        short sb_flags;   /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
#define    SB_LOCK 0x01 /* lock on data queue (so_rcv only) */
#define    SB_WANT0x02 /* someone is waiting to lock */
#define    SB_WAIT  0x04 /* someone is waiting for data/space */
#define    SB_SEL     0x08 /* buffer is selected */
#define    SB_COLL 0x10 /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

6.1.3. Socket Connection Queueing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two sides are considered distinct. One side is termed *active*, and generates connection requests. The other side is called *passive* and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transfered to *so_q*, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

6.2. Protocol Layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short pr_type;          /* socket type used for */
        short pr_family;        /* protocol family */
        short pr_protocol;              /* protocol number */
        short pr_flags;         /* socket visible attributes */
/* protocol-protocol hooks */
        int     (*pr_input)();          /* input to protocol (from below) */
        int     (*pr_output)();         /* output to protocol (from above) */
        int     (*pr_ctlinput)(); /* control input (from below) */
        int     (*pr_ctloutput)();      /* control output (from above) */
/* user-protocol hook */
        int     (*pr_usrreq)();         /* user request */
/* utility hooks */
        int     (*pr_init)();           /* initialization routine */
        int     (*pr_fasttimo)(); /* fast timeout (200ms) */
        int     (*pr_slowtimo)();/* slow timeout (500ms) */
        int     (*pr_drain)();          /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_userreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define     PR_ATOMIC    0x01        /* exchange atomic messages only */
#define     PR_ADDR 0x02        /* addresses given with messages */
#define     PR_CONNREQUIRED      0x04        /* connection required by protocol */
#define     PR_WANTRCVD      0x08        /* want PRU_RCVD calls */
#define     PR_RIGHTS    0x10        /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

6.3. Network-Interface Layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
struct ifnet {
        char  *if_name;          /* name, e.g. "en" or "lo" */
        short if_unit;           /* sub-unit for lower level driver */
        short if_mtu;                    /* maximum transmission unit */
        int   if_net;                    /* network number of interface */
        short if_flags;          /* up/down, broadcast, etc. */
        short if_timer;          /* time 'til if_watchdog called */
        int   if_host[2];        /* local net host number */
        struct      sockaddr if_addr;       /* address of interface */
        union {
              struct      sockaddr ifu_broadaddr;
              struct      sockaddr ifu_dstaddr;
        } if_ifu;
        struct      ifqueue if_snd;         /* output queue */
        int   (*if_init)();      /* init routine */
        int   (*if_output)();            /* output routine */
        int   (*if_ioctl)();             /* ioctl routine */
        int   (*if_reset)();             /* bus reset routine */
        int   (*if_watchdog)();/* timer routine */
        int   if_ipackets;       /* packets received on interface */
        int   if_ierrors;        /* input errors on interface */
        int   if_opackets;               /* packets sent on interface */
        int   if_oerrors;        /* output errors on interface */
        int   if_collisions;             /* collisions on csma interfaces */
        struct      ifnet *if_next;
};
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define   IFF_UP    0x1  /* interface is up */
#define   IFF_BROADCAST   0x2  /* broadcast address valid */
#define   IFF_DEBUG    0x4  /* turn on debugging */
#define   IFF_ROUTE    0x8  /* routing entry installed */
#define   IFF_POINTOPOINT 0x10 /* interface is point-to-point link */
#define   IFF_NOTRAILERS  0x20 /* avoid use of trailers */
#define   IFF_RUNNING 0x40 /* resources allocated */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets.*

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

* *Trailer* protocols are normally disabled on the Sun Workstation.

7. Socket/Protocol Interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define    PRU_ATTACH 0     /* attach protocol */
#define    PRU_DETACH 1     /* detach protocol */
#define    PRU_BIND      2  /* bind socket to address */
#define    PRU_LISTEN   3   /* listen for connection */
#define    PRU_CONNECT      4     /* establish connection to peer */
#define    PRU_ACCEPT  5    /* accept connection from peer */
#define    PRU_DISCONNECT 6       /* disconnect from peer */
#define    PRU_SHUTDOWN     7     /* won't send any more data */
#define    PRU_RCVD      8  /* have taken data; more room now */
#define    PRU_SEND      9  /* send this data */
#define    PRU_ABORT    10  /* abort (fast DISCONNECT, DETATCH) */
#define    PRU_CONTROL      11    /* control operations on protocol */
#define    PRU_SENSE     12 /* return status into m */
#define    PRU_RCVOOB 13    /* retrieve out of band data */
#define    PRU_SENDOOB      14    /* send out of band data */
#define    PRU_SOCKADDR     15    /* fetch socket's address */
#define    PRU_PEERADDR     16    /* fetch peer's address */
#define    PRU_CONNECT2     17    /* connect two sockets */
/* begin for protocols internal use */
#define    PRU_FASTTIMO     18    /* 200ms timeout */
#define    PRU_SLOWTIMO     19    /* 500ms timeout */
#define    PRU_PROTORCV     20    /* receive from below */
#define    PRU_PROTOSEND    21    /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(up, req, m, addr, rights);
int error; struct socket *up; int req; struct mbuf *m, *rights; caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH

When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH

This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND

When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN
The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to accept a connection.

PRU_CONNECT
The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting,* it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT
Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT
Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN
This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD
This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND
Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT
This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL
The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be

obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE
: The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics, protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB
: Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB
: Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR
: The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR
: The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2
: The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the *socketpair*(2) system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO
: A "fast timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO
: A "slow timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV
: This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND
: This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/Protocol Interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, "*in*ternet *p*rotocol *c*ontrol *b*lock", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

```
error = ip_output(m, opt, ro, allowbroadcast);
int error; struct mbuf *m, *opt; struct route *ro; int allowbroadcast;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, *allowbroadcast* is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m);
struct mbuf *m;
```

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

```
        (void) (*protosw[].pr_ctlinput)(req, info);
        int req; caddr_t info;
```

The *req* parameter is one of the following,

```
#define PRC_IFDOWN              0       /* interface transition */
#define PRC_ROUTEDEAD           1       /* select new route if possible */
#define PRC_QUENCH              4       /* some said to slow down */
#define PRC_HOSTDEAD            6       /* normally from IMP */
#define PRC_HOSTUNREACH         7       /* ditto */
#define PRC_UNREACH_NET         8       /* no route to network */
#define PRC_UNREACH_HOST        9       /* no route to host */
#define PRC_UNREACH_PROTOCOL    10/* dst says bad protocol */
#define PRC_UNREACH_PORT        11      /* bad port # */
#define PRC_MSGSIZE             12      /* message size forced drop */
#define PRC_REDIRECT_NET        13      /* net routing redirect */
#define PRC_REDIRECT_HOST       14      /* host routing redirect */
#define PRC_TIMXCEED_INTRANS    17/* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS      18      /* lifetime expired on reass q */
#define PRC_PARAMPROB           19      /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

9. Protocol/Network-Interface Interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

9.1. Packet Transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

9.2. Packet Reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
: This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
: This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)
: This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and Routing Issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing Tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long    rt_hash;         /* hash key for lookups */
        struct    sockaddr rt_dst; /* destination net or host */
        struct    sockaddr rt_gateway;  /* forwarding agent */
        short rt_flags;          /* see below */
        short rt_refcnt;         /* no. of references to structure */
        u_long    rt_use;                  /* packets sent using route */
        struct    ifnet *rt_ifp;           /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the *rt_gateway* field instead of *rt_dst*, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed *equal* in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using *netstat*(1).

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing Table Interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

10.3. User-Level Routing Policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

11. Raw Sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control Blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct     rawcb *rcb_next;        /* doubly linked list */
        struct     rawcb *rcb_prev;
        struct     socket *rcb_socket;     /* back pointer to socket */
        struct     sockaddr rcb_faddr;     /* destination address */
        struct     sockaddr rcb_laddr;     /* socket's address */
        caddr_t    rcb_pcb;                /* protocol specific stuff */
        short      rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

11.2. Input Processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct      sockproto raw_proto;
        struct      sockaddr raw_dst;
        struct      sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output Processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and Congestion Control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for 'normal' network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be 'turned off' as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory Management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 16 fitting in a 2048 byte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol Buffering Policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the 'silly window syndrome' described in [Clark82] at both the sending and receiving ends.

12.3. Queue Limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a 'defensive' mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable 'packet handling rate' can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet Forwarding

When packets can not be forwarded because of memory limitations, the system generates a 'source quench' message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a 'routing loop' resulted in network saturation and every host on the network crashing.

13. Out of Band Data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols prerogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

Appendix A. Acknowledgements and References

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81].

Appendix B. References

[Boggs79]	Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.
[BBN78]	Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP.* BBN Technical Report 1822. May 1978.
[Cerf78]	Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.
[Clark82]	Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.
[DEC80]	Digital Equipment Corporation; *DECnet DIGITAL Network Architecture - General Description.* Order No. AA-K179A-TK. October 1980.
[Gurwitz81]	Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.
[ISO81]	International Organization for Standardization. *ISO Open Systems Interconnection - Basic Reference Model.* ISO/TC 97/SC 16 N 719. August 1981.
[Joy82a]	Joy, W.; Cooper, E.; Fabry, R.; Leffler, S.; and McKusick, M.; *System Interface Overview.* Computer Systems Research Group, Technical Report 5. University of California, Berkeley. Draft of September 1, 1982.
[Postel79]	Postel, J., ed. *DOD Standard User Datagram Protocol.* Internet Working Group, IEN 88. May 1979.
[Postel80a]	Postel, J., ed. *DOD Standard Internet Protocol.* Internet Working Group, IEN 128. January 1980.
[Postel80b]	Postel, J., ed. *DOD Standard Transmission Control Protocol.* Internet Working Group, IEN 129. January 1980.
[Xerox81]	Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.
[Zimmermann80]	Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.

Writing Device Drivers for the Sun Workstation

Table of Contents

List of Tables

Writing Device Drivers for the Sun Workstation

This document is a guide to adding software drivers for new devices to the kernel.

One of the UNIX† Operating System's major services to application software is a device-independent view of the hardware that stores and retrieves data and communicates with the outside world. The interface between UNIX application software and a given piece of raw hardware is provided by a *device driver* for that piece of hardware. A device driver provides an interface between the UNIX operating system's device-independent scheme of things and the special characteristics of a particular piece of hardware.

1. Introduction

The kernel supplied with the Sun system is a *configurable kernel,* meaning that it is possible (within limits) to make changes to the kernel and to add new device driver modules. A detailed explanation of how to configure and build a kernel is in *Building UNIX Systems with Config* in the *System Manager's Manual.*

This document is aimed at the Sun user who has some expertise in writing UNIX device drivers, and who wishes to connect a new Multibus device to the Sun system. The UNIX system that runs on the Sun Workstation supports several different types of devices, and the scope of this document is limited to writing device drivers for the kinds of devices not already supplied by Sun. If you have no previous experience writing UNIX device drivers, you should expect to seek some advice from the Sun technical support organization or an outside consultant experienced in writing UNIX drivers. We can classify devices and their drivers into seven major categories:

1. Co-processors.
2. Disks and tapes.
3. Network interface drivers such as Ethernet or X.25.
4. Serial communications multiplexors.
5. General DMA devices such as driver boards for raster-oriented printers or plotters.
6. Programmed I/O devices.
7. Frame buffers.

This manual *only* addresses devices and drivers in categories 5, 6, and 7. There is a wide range of devices which Sun does not support for which you might want to write a device driver. This document is primarily concerned with creating device drivers for devices such as parallel interfaces, analog to digital (A/D) converters, digital to analog (D/A) converters, interfaces to special outboard processors, frame buffers, memory-mapped graphics boards, and so on. Such devices can be cast into the model of *unstructured* or *character* I/O devices in the UNIX I/O system

† UNIX is a trademark of Bell Laboratories.

scheme, as opposed to block I/O devices that support a UNIX file system. Character I/O devices may support *read* and *write* operations, and may provide an *ioctl* interface for controlling the devices. Such devices may also provide for being mapped into the user's virual address space by supporting the *mmap* system call.

This document *does not* address devices and drivers in categories 1 thru 4. In particular, the considerations in writing device drivers for disks, tapes, serial communication devices, and local network interface drivers are quite involved — we do not discuss the construction of such drivers in this document. Most Sun customers should find that the extensive use of standards in the Sun product line should allow them to use hardware interfaces already provided by Sun to drive such peripherals.

To add a new hardwate device-controller and its device driver to the system you must:

1. Get the device controller hardware into a state where you know it works as advertised — it is *extremely* difficult to debug your device driver software (step 4 below) if the hardware is not known to be working,
2. Write the device driver itself,
3. Add it to the system configurator's data base, describe a system containing the driver, and compile this system containing the new device driver,
4. Debug the driver.

Chapter 2 is a general overview of the hardware and software environment provided by the Sun Workstation.

Chapter 3 is a description of the I/O system and device drivers. Chapter 3 provides a model of a very simple device driver and describes the issues involved in programming device drivers on the Sun system.

Chapter 4 is a description of how to add a new device driver to the kernel.

Finally, samples of actual drivers are included with this document so that the reader can see how the actual code is used. The drivers we have included as samples are:

cgone	A simple memory-mapped driver for the black and white framebuffer.
sky	A simple programmed I/O driver for the SKY floating-point board.
vp	A DMA device driver for the Versatec printer/plotter.

Hint: Spend as much time as you need in the Sun Workstation PROM monitor poking, prodding and cajoling your device until you are thoroughly familiar with its behavior. This will save you a lot of grief later. There is a discussion a little later on the kinds of things you can do with the PROM monitor.

2. General Hardware and Software Topics

2.1. Device Names and Device Numbers — The /dev Directory

All devices and special files are defined externally in the */dev* directory. Devices are characterized by a major device number, a minor device number, and a class (block or character). When a file of any type is opened, the device driver to call is obtained from the entry in the */dev* directory. Entries in the */dev* directory are created via the *mknod*(8) (make a node) administration command. Here is a fragment of what the */dev* directory looks like from an **ls –l** command:

Table 1: A sample listing of the /dev Directory

Type	*permissions*	*size*	*owner*	*major #*	*minor #*	*date*	*name*
c	rw--w--w-	1	henry	0,	0	Feb 21 09:45	console
c	rw-r--r--	1	root	3,	1	Dec 28 16:18	kmem
c	rw-------	1	root	3,	4	Jan 13 23:07	mbio
c	rw-------	1	root	3,	3	Jan 13 23:07	mbmem
c	rw-r--r--	1	root	3,	0	Dec 28 16:18	mem
c	rw-rw-rw-	1	root	13,	0	Dec 28 16:18	mouse
c	rw-rw-rw-	1	root	3,	2	Feb 22 16:40	null
c	rw-------	1	root	9,	0	Dec 28 16:19	rxy0a
c	rw-------	1	root	9,	1	Dec 28 16:19	rxy0b
							. . .
c	rw-------	1	root	9,	6	Feb 25 1984	rxy0g
c	rw-------	1	root	9,	7	Dec 28 16:19	rxy0h
b	rw-------	1	root	3,	0	Feb 25 1984	xy0a
b	rw-------	1	root	3,	1	Jan 17 20:12	xy0b
							. . .
b	rw-------	1	root	3,	6	Dec 28 16:19	xy0g
b	rw-------	1	root	3,	7	Dec 28 16:19	xy0h

The connection between the specific device name in the */dev* directory is made through two C structures named *bdevsw* (block device switch table) and *cdevsw* (character device switch table) in the file called *conf.c*. When you add a new device driver you must add entries to the corresponding structure. Since we are discussing only character-oriented devices in this manual, you can ignore the *bdevsw* structure and concentrate on the *cdevsw* structure.

Application programs make calls upon the operating system to perform services such as opening a file, closing a file, reading data from a file, writing data to a file, and other operations that are done in terms of the file interface. The operating system code turns these requests into specific requests on the device driver involved with that particular file. The glue between the specific file operation involved and the device driver entry-point is through the *bdevsw* and *cdevsw* tables.

Entries in *bdevsw* or *cdevsw* contain an array of entry points into the device drivers. The position in the structure corresponds to the major device number assigned to the device. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The *cdevsw* table specifies the interface routines present for character devices. Each character device may provide seven functions: *open*, *close*, *read*, *write*, *ioctl*, *select*, and *mmap*. If a call on the routine should be ignored, (for example *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev*; if it should be considered an error, (for example *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the array of *tty* structures associated with the driver.

Here is what the declaration of the character device switch looks like. Each entry (row) is the only link between the main unix code and the driver. The initialization of the device switches is in the file *conf.c*.

```
struct cdevsw
{
int   (*d_open)();       /* routine to call to open the device */
int   (*d_close)();      /* routine to call to close the device */
int   (*d_read)();       /* routine to call to read from the device */
int   (*d_write)();      /* routine to call to write to the device */
int   (*d_ioctl)();      /* special interface routine */
int   (*d_stop)();       /* routine to call to open the device */
int   (*d_reset)();      /* routine to call to open the device */
struct tty *d_ttys;      /* tty structure */
int   (*d_select)();     /* routine to call to select the device */
int   (*d_mmap)();       /* routine to call to mmap the device */
};
```

Only the console driver uses the *tty* structure. All other devices set this field to zero.

And here is a typical line from the *conf.c* file which fills in the requisite pointers in the *cdevsw* structure:

```
        .
        .
        .
        All the other cdevsw entries between 0 and 13 appear first
   {
cgoneopen, cgoneclose, nodev,      nodev,      /*14*/
cgoneioctl, nodev,     nodev,      0,
seltrue,    cgonemmap,
   },

        Then all the other cdevsw entries from 15 upwards
        .
        .
        .
```

In the Sun system, a number of devices in *cdevsw* are preassigned. The table below shows the assignments to date. Those major device numbers shown as 'For Local Use' are available for user-written device drivers.

Table 2: Character Device Number Assignments

Character-Device Number Assignments		
Major Device Number	*Device Abbreviation*	*Device Description*
0	cn	Sun Console
1	oct	Central Data Octal Board
2	sy	Indirect TTY
3	Not available	
4	ip	Raw Interphase Disk Device
5	tm	Raw Tapemaster Tape Device
6	vp	Ikon Versatec Parallel Controller
7	Not available	
8	ar	Archive Tape Controller
9	xy	Raw Xylogics Disk Device
10	mti	Systech MTI
11	unused	no device
12	zs	Sun-2 UARTS
13	ms	Mouse
14	cg	Color Graphics Board
15	win	Window Pseudo Device
16	ii	INGRES lock device
17	sd	SCSI disk
18	st	SCSI tape
19	nd	Raw Network Disk Device
20	pts	Pseudo TTY
21	ptc	Pseudo TTY
22	bw	Monochrome Video board
23	ropc	RasterOp Chip
24	sky	no device
25	pi	Parallel input device
26	bwone	Sun-1 Monochrome frame buffer
27	bwtwo	Sun-2 Monochrome frame buffer
28	vpc	Parallel driver for Versatec printer
29	kbd	Sun keyboard driver
30–??	For Local Use	

2.2. The Sun Hardware and the Multibus

The Sun system hardware is built around the IEEE-P796 Multibus. This section discusses several issues relevant to the Multibus and devices that can be obtained for it.

2.2.1. Multibus Memory Address Space and I/O Address Space

Although Sun uses Motorola MC68000 family processors for its products, the systems are actually built around the IEEE-P796 Multibus. The MC68000 processors do what is known as 'memory-mapped' input-output in that you just store data somewhere or fetch data from somewhere to transfer data to or from a peripheral device or memory — there is no distinction between the memory and peripherals. The Multibus, on the other hand, was originally designed for processors that have one kind of instruction for storing data in memory or fetching data from memory (instructions such as MOV), and a different kind of instruction (such as IN and OUT) for transferring data to or from peripheral devices. Thus the Multibus has the notion of two separate address spaces:

Multibus memory space
: is simply used for memory or devices that look like memory, in that you talk to such devices simply by writing data to memory locations or reading from memory locations. The Sun color controller board is a good example of a device that is addressed as memory in the Multibus memory address space. Devices that look like memory are called 'memory mapped' devices.

Multibus I/O address space
: is another 'space' that is typically used for device control registers. Devices using the I/O address space are said to be 'I/O mapped' devices.

This concept of two different address spaces derives from the Intel 8080 family of processors. The MC68000 family doesn't have this separation of memory and I/O, but treats the entire universe as one address space. The Sun memory management hardware can map any portion of the system's address space to the Multibus memory space or the Multibus I/O address space. Ultimately, the different kinds of address-space end up just waggling different control lines on the Multibus.

Be aware though, that the memory space of the Multibus is designed for a 20-bit r or a 24-bit addressing scheme (Sun uses 20-bit addresses), whereas the I/O space of the Multibus is only an 8-bit or a 16-bit addressing scheme (Sun uses 16-bit addresses), and some older Multibus boards only accept 8-bit I/O addresses.

2.2.2. Byte Ordering Issues

The Sun processor is a Motorola MC68000 processor, built on an IEEE-P796 Multibus board. IEEE-P796 and Motorola do not agree on the addressing of bytes in a word. IEEE-P796 and Motorola both agree that there are 16 bits in a word and that is about all they agree on. The disagreement about which end of the word contains byte 0 leads us into two separate problems, with two separate fixes you must apply:

1. You are moving a single *byte* across the interface between the MC68000 and the P796 Bus. Because of the disagreement about which end of the word the byte actually appears in, you have to toggle the least significant bit of the *byte* address.
2. You are moving a whole 16-bit *word* across the interface between the MC68000 and the P796 Bus. This word actually contains a byte structure destined for the device on the other side of the bus. The device will interpret the byte-order different from what you thought, and so in this case you must physically swap the bytes in the word before you ship the word across the bus interface.

Here are a few pictures describing the problem in detail:

Motorola Byte Ordering

bit 15	bit 0
Byte 0	**Byte 1**

IEEE-P796 Byte Ordering

bit 15	bit 0
Byte 1	**Byte 0**

That is, Motorola places byte 0 in bits 8 thru 15 of the word, whereas IEEE-P796 places byte 1 in those bits. The only place where this causes trouble is when you are moving a single *byte* across the interface between the MC68000 and the Multibus. If you did everything with the 68000, or everything on the Multibus, there would never be any conflict, since things would be consistent. However, as soon as you cross the boundary between them, the byte order is reversed. What this means in practice is that you have to toggle the least significant bit of the address of any *byte* destined for the Multibus.

To clarify this, consider an interface for a hypothetical Multibus board containing only two 8-bit I/O registers, namely a control and status register (csr) and a data register (we actually use this design later on in our example of a simple device driver). In this board, we place the command and status register at Multibus byte location 600, and the data register at Multibus byte location 601. The Multibus picture of that device looks like this:

Hypothetical Board Registers

bit 15	bit 0
Location 601 **DATA**	**Location 600** **CSR**

But the 68000 processor views that device as looking like this:

Hypothetical Board Registers

bit 15	bit 0
Location 600 **CSR**	**Location 601** **DATA**

so that if you were to read location 600 from the point of view of the 68000 processor, you'd really end up reading the DATA register off the Multibus instead. So, when we define the *skdevice* data structure for that board, we define it like this:

```
struct skdevice {
        char sk_data;   /* 01: Data Register */
        char sk_csr;    /* 00: command(w) and status(r) */
};
```

This rule (flipping the least significant bit of the address) holds good for all *byte* transfers which cross the line between the MC68000 and the Multibus.

Take special care when a Multibus device structure contains mixed bytes and words. Many of the Multibus device controllers on the market are geared up for the 8-bit 8080 and Z80 style chips, and don't understand 16-bit data transfers. Because of this, such controllers are quite happy to place what is really a word quantity (such as a 16-bit address which must be two-byte aligned in the MC68000) starting on an odd byte boundary. Some of the device drivers use 16-bit or 20-bit addresses (many don't know about 24-bit addresses), and it often happens that you have to chop an address into bytes by shifting and masking, and assign the halves or thirds of the address one at a time, because the device controller wants to place word-aligned quantities on odd byte boundaries. Note also that many Multibus boards are geared up for the 8086 family with its segmented adress scheme. An 8086 (20-bit) address really consists of a 4-bit segment number and a 16-bit address. You usually have to deal with the 4-bit part and the 16-bit part separately. For a good example of what we're talking about here, look at the code for *vp.c*, (attached as an appendix to this document).

2.2.3. Things to Watch for in Multibus Boards

Although there are a myriad of vendors offering Multibus products, be aware that the Multibus is a 'standard' that evolved from a bus for 8-bit systems to a bus for 16-bit systems. Read vendors' product literature *carefully* (especially the fine print) when selecting a Multibus board. The memory address space of the Multibus is *supposed to be* 20 bits wide or 24 bits wide and the I/O address space of the Multibus is *supposed to be* 16 bits wide. In practice, some older boards are limited to 16 bits of address space and only 8 bits of I/O space. In particular, watch for the following things:

- For a memory-mapped board, ensure that the board can actually handle a full twenty bits of addressing. Older Multibus boards often can only handle sixteen address lines. The Sun system assumes there is a 20-bit Multibus memory space out there. If the Multibus board you're talking at can only handle 16-bit addresses, it will ignore the upper four address lines, and this means that such a board 'wraps around' every 64K, which means that in our system, the addresses that such a board responds to would be replicated sixteen times through the one-Megabyte address space on the Multibus.
- A memory-mapped Multibus board that uses 24-bit addressing (thereby using the P2 bus on the backplane) must use a P2 bus that is physically isolated from the P2 bus that any Sun boards use. See the *Sun Configuration Guide* for information on configuring boards in the backplane.
- For an I/O-mapped board (one that uses I/O registers), make sure that the board can handle 16-bit I/O addressing. Some older boards can't cope and only use eight-bit I/O addressing. In our system, the address spaces of such boards would find themselves replicated every 256 bytes in the I/O address space. Trying to fit such a board into the Sun System would severely curtail the number of I/O addresses available in the system.
- Watch out for boards containing PROM code that expects to find a CPU busmaster with an Intel 8080, 8085, or 8086 on it. Such boards are of course useless in the Sun System.

- Take special care to determine how the board generates interrupts. A board should put up an interrupt when the device it is controlling is ready for more data *and* the board is ready for more data — we have experienced designs where the interrupt indicated that the board was ready, or the device was ready, but not both at once. A board should ideally come up in its power up state with interrupts disabled and only start interrupting when told to. There should also be a way to determine that a board has actually generated an interrupt. Finally, an interrupting board should shut off its interrupt when it is told to.

2.3. DMA Devices

Many device controller boards are capable of what is known as Direct Memory Access or DMA. This means that the processor tells the device controller the address in memory where a data transfer is to take place, plus the length of the data transfer, and then tells the device controller to start the transfer. The data transfer then takes place without further intervention on the part of the processor. When the transfer is complete, the device controller interrupts to say that the transfer is finished.

2.3.1. Sun Multibus DVMA

Direct Virtual Memory Access (DVMA) is a mechanism provided by the Sun memory management unit that allows DMA from devices on the Multibus to Sun processor memory, or from Multibus master devices directly to Multibus slaves without going through processor memory. DVMA uses the first 256K bytes of the Multibus address memory address space to map addresses between Sun processor memory and the Multibus memory address space.

On the Sun-2, the memory management unit is always listening to the Multibus for memory references. When a request to read or write Multibus memory between addresses 0 and 256K comes up, the DVMA hardware takes the address, adds 0xF00000 to it,[1] and goes through the kernel memory map to find the location in processor memory that will be used. Thus if you wish to do DMA over the Multibus, you must make the appropriate entries in the kernel memory map. As you might expect, there are functions to help with this chore.

2.4. Allocation of Multibus Memory and I/O in the Sun System

Here are some simple rules for the way that Multibus memory resources are doled out in the Sun system.

No devices may be assigned addresses below 256K in Multibus memory space; the CPU uses these addresses for DVMA.

Devices that interface to the Sun system do so either through I/O registers in Multibus address space, or through the Multibus memory space. In some cases, a device may have both I/O registers and memory on the Multibus. The Sun system makes the assumption that any address lower than 64K is a Multibus I/O address. This is a reasonable assumption given that user-installed Multibus memory cannot appear in this region of the address space anyway. This assumption is carried through into the autoconfiguration routines in that addresses less than 64K are automatically mapped to the Multibus I/O address space.

[1] The system places the Multibus memory address space at location 0xF00000 in the virtual address space.

To configure such a device,

1. the *probe* function for the device driver must return the amount of Multibus memory space that the device uses,
2. Multibus I/O address space is at 'mbio' and may be addressed as such,
3. the autoconfiguration utility (config) can not deal with I/O address space at the same time as memory address space for the same device.

The table on the next page shows a map of how Multibus memory is laid out in the Sun system.

Table 3: Sun-2 Multibus Memory Map

Address	Device
0x00000	DVMA Space
	.
	. (256 Kbytes)
	.
0x3f800	DVMA Space
0x40000	Sun Ethernet Memory (#1)
	.
	. (256 Kbytes)
	.
0x7f800	Sun Ethernet Memory (#1)
0x80000	SCSI (#1)
	.
	. (16 Kbytes)
	.
0x83800	SCSI (#1)
0x84000	SCSI (#2)
	.
	. (16 Kbytes)
	.
0x87800	SCSI (#2)
0x88000	Sun Ethernet Control Info (#1)
	.
	. (16 Kbytes)
	.
0x8b800	Sun Ethernet Control Info (#1)
0x8c000	Sun Ethernet Control Info (#2)
	.
	. (16 Kbytes)
	.
0x8f800	Sun Ethernet Control Info (#2)
0x90000	*** FREE ***
	.
	. (64 Kbytes
	.
0x9f800	*** FREE ***

Address	Device
0xa0000	Sun Ethernet Memory (#2)
	.
	. (64 Kbytes)
	.
0xaf800	Sun Ethernet Memory (#2)
0xb0000	*** **FREE** ***
	.
	. (64 Kbytes)
	.
0xbf800	*** **FREE** ***
0xc0000	Sun Model 100 or Model 150 Frame Buffer
	.
	. (128 Kbytes)
	.
0xdf800	Sun Model 100 or Model 150 Frame Buffer
0xe0000	3COM Ethernet (#1)
0xe0800	3COM Ethernet (#1)
0xe1000	3COM Ethernet (#1)
0xe1800	3COM Ethernet (#1)
0xe2000	3COM Ethernet (#2)
0xe2800	3COM Ethernet (#2)
0xe3000	3COM Ethernet (#2)
0xe3800	3COM Ethernet (#2)
0xe4000	*** **FREE** ***
	.
	. (16 Kbytes)
	.
0xe7c00	*** **FREE** ***
0xe8000	Sun Color
	.
	. (64 Kbytes)
	.
0xf7800	Sun Color
0xf8000	*** **FREE** ***
	.
	. (16 Kbytes)
	.
0xff800	*** **FREE** ***

2.5. Multibus Resource Management

The following data structures in fact reflect the layout of information in the configuration file which we describe in a later part of this paper. Controllers and devices can be thought of as being attached to the Multibus Certain kinds of devices (disks and tapes) are then thought of as being slaves to their controllers. This layout gives rise to three data structures whose descriptions exist in the header file */usr/include/sundev/mbvar.h*.

Multibus The first data structure is the Multibus header data structure. The fact that it is called 'Multibus' is a complete red herring — it is simply a hook to hang all the other data structures on. The Multibus data structure contains a list of controllers using this resource.

Controller Contains a list of structures that describe controllers. There is sometimes considerable confusion as to exactly what is a controller and what is a device. Essentially a *controller* is a piece of hardware that can control more than one device, but *only one data transfer can be active at a time.* Each device controller on the Multibus has a structure associated with it. The structure is called *mb_ctlr* and can be found in */usr/include/sundev/mbvar.h.*

Device Contains a list of devices. Each device driver has a data structure describing how the Multibus resource-management routines view the driver. The per-driver data structure is called *mb_driver* and can be found in */usr/include/sundev/mbvar.h.* The device data structures are either hooked directly onto the Multibus header structure, or they are hooked to controller structures in which case the devices are said to be *slaves* to their controllers. The device structure, *mb_driver,* is the really important data structure that you need to be concerned with when writing a driver. Here is the layout of the *mb_driver* structure:

```
struct mb_driver {
        int    (*mdr_probe)();  /* see if a driver is really there */
        int    (*mdr_slave)();  /* see if a slave is there */
        int    (*mdr_attach)(); /* setup driver for a slave */
        int    (*mdr_go)();     /* routine to start transfer */
        int    (*mdr_done)();   /* routine to finish transfer */
        int    (*mdr_intr)();   /* interrupt routine */
        u_long      *mdr_ioaddr;    /* device csr addresses */
        u_long      *mdr_maddr;     /* device memory address */
        int    mdr_size;  /* amount of memory space needed */
        char  *mdr_dname;     /* name of a device */
        struct      mb_device **mdr_dinfo;     /* backpointers to mbdinit structs */
        char  *mdr_cname;     /* name of a controller */
        struct      mb_ctlr **mdr_cinfo; /* backpointers to mbcinit structs */
        short mdr_flags; /* want exclusive use of Multibus */
        struct      mb_driver  *mdr_link; /* interrupt routine linked list */
};
```

Here is a brief discussion of the fields in the *mb_driver* structure and what parts of it you need to fill in when declaring *mb_driver*:

mdr_probe
is a pointer to a *probe* function within your driver. *Probe* determines if the device for which this driver is written is really there in the system. Fill in this field only if your driver has a *probe* routine (it generally will).

mdr_slave
is a pointer to a *slave* function within your driver. Fill in this field for controllers that have more than one device. The *slave* function always returns a 1.

mdr_attach
is a pointer to an *attach* function within your driver. The *attach* function does preliminary setup work for a slave device. Typical applications include reading the label from a disk. Fill in this field only if there is an *attach* routine in your driver. In general, the drivers we are considering in this paper don't have *attach* routines, and so you fill in a zero (0) in this field.

mdr_go
mdr_done
are pointers to *go* and *done* functions within your driver. These fields are usually zero for the types of drivers we talk about in this paper. They are normally for disk drivers who can't afford to wait for *mbsetup*.

mdr_intr
is a pointer to an interrupt routine (function) within your driver. Fill in this field if your driver actuall has an interrupt routine (in general it should). If your driver doesn't have an interrupt routine, fill in a zero (0) in this field.

mdr_ioaddr
points to an array of unsigned long's declared within your driver. The array contains the address(es) of the device in Multibus I/O space. The last entry in the array should be a zero. If your device actually exists in Multibus I/O space, you must fill in this field with the name of the array in your driver, otherwise place a zero (0) here. The system uses the array of addresses as parameters to the driver's *probe* function at system startup time.

mdr_maddr
points to an array of unsigned long's declared within your driver. The array contains the address(es) of the device in Multibus Memory space. The last entry in the array should be a zero. If your device actually exists in Multibus Memory space, you must fill in this field with the name of the array in your driver, otherwise place a zero (0) here. The system uses the array of addresses as parameters to the driver's *probe* function at system startup time.

mdr_size
is the size in bytes of the amount of memory that a memory-mapped device requires. This field *must* be filled in if *mdr_maddr* is used for a memory-mapped device.

mdr_dname
is the name of the device for which this driver is written. This field takes the form of a regular null-terminated C string.

mdr_dinfo
an array of pointers to *mb_device* structures. Auto configuration fills in the pointers, then the driver can access *mb_device* structures if it wants to.

mdr_cname
is the name of the controller for which this driver is written. This field takes the form of a regular null-terminated C string. Fill in this field if you actually have a controller.

mdr_cinfo
an array of pointers to *mb_controller* structures. Auto configuration fills in the pointers, then the driver can access *mb_controller* structures if it wants to.

mdr_flags
consists of some flags, as follows:

MDR_XCLU
: needs exclusive use of bus

MDR_DMA
: device does Multibus DMA

MDR_SWAB
: Multibus buffer must be swabbed

MDR_OBIO
: device in on-board I/O space

These flags must be OR'ed together if you wish to place any of that information there. Place a zero (0) in this field if none of the flags apply to this driver.

mdr_link

This field is used by the autoconfiguration routines and is not for the driver's use.

2.6. Getting the Board Working and in a Known State

This section discusses getting the hardware device controller operational and in a known state,

Before you even *think* about writing any code you should check out the Multibus board by performing various tests.

First, make sure that the board is properly set up as defined in the vendor's manual. Things you have to select in general are:

- I/O register addresses for those boards that use I/O ports on the Multibus,
- Memory base address for those boards that use memory space on the Multibus,
- Interrupt level selection.

Then, take your system down and power it off. Plug your Multibus board into the card cage and attempt to bring the system back up. If you cannot boot the system, then there is a problem such as the board not really working or the board responding at an address used by other boards in the system. You must resolve this problem before proceeding further.

Next take your system down again and see if the device responds. from the monitor, try some of the following things:

- Try reading from the board status register(s) if there are any.
- Try writing to the board control or data register(s) if there are any. Then try reading the data back to see if it got written properly (assuming that the board can read back what you wrote).
- Try sending data to the actual device itself through the board if this is possible.
- Switch the actual device offline and online and watch the status bits go on and off (if this is possible).

For example, if you have a line printer, try to print a line with a few characters. Be aware that bit and byte ordering issues are critical in The section just below on *Using the Sun CPU PROM Monitor* has some hints on reading and writing device registers. Be aware that bit and byte ordering issues are critical in this process; the main reason for doing this step is to discover what the board really does. When you have developed confidence in how the board works you can proceed to write a driver for it.

2.6.1. Using the Sun CPU PROM Monitor

To do some of the poking around as described in the previous paragraphs, you can use the CPU PROM monitor whose commands are described in detail in the *System Internals Manual*. The PROM monitor has commands for looking at memory locations. So if you have located your new Multibus board at a specific place in the address space, you could use the monitor to look at that place to see if there's anything there. For example, if you think your board has an I/O control register at location 0x600, you could use the monitor's 'open a byte location' command to look at that place in memory:

```
> o eb0600
```

and so on. If you get a bus error timeout, the board isn't there and you have to go back to the manual to see if you've set the address jumpers correctly.

Here are a couple of notes about using the monitor to look at devices. When you use the Monitor's 'o' command to open a location, the Monitor *reads* the contents of that location and displays them before asking you what you want to put there (if anything). Now some devices (the Intel 8251A and the Signetics 2651 immediately spring to mind) use the same location (register) to address *two* separate internal mode registers, and the chip has internal state-logic that sequences around them in 1-2-1-2... order. So suppose you want to put something in mode register 1 of the 8251? You open that location, the Monitor displays the contents, and you then write the byte. Being cautious, you then open that location again and bingo! the data you wrote isn't there — it's in the second register because the action of *reading* that location sequenced you on to the second register. To do this thing right you have to use the Monitor's 'write without looking' facility and then read the locations back later to check.

Another chip that has internal sequencing logic of this type is the NEC PD7201 PCC. This chip has a a bunch of internal data registers. You load data-register 0 with the number of the data register into which the next byte of data will go, then you send the byte of data and it goes into that specific data register, and then you are back to data-register 0 again, all done with internal sequencing logic.

Another chip of a similar ilk is the AMD 9513 timer. This chip has a data pointer register for pointing at the data-register into which a data byte will go. When you send a byte to the data register, the pointer gets incremented. The design of the chip is such that you *can't read the pointer register to find out what's in it!*

3. Device Drivers

This section discusses the major issues in creating a device driver for the system.

A first step in writing a device driver is deciding what sort of interface the device should provide to the system. The way in which *read* and *write* operations should occur, the kinds of control operations provided via *ioctl*, and whether the device can be mapped into the user's address space using the *mmap* system call, should be decided early in the process of designing the driver.

Device drivers have access to the memory management and interrupt handling facilities of the UNIX system. The device driver is called each time the user program issues an *open*, *close*, *read*, *write*, *mmap*, or *ioctl* system call. The device driver can arrange for I/O to happen synchronously, or it can allocate buffers so that output can proceed while the user process runs, or gather input while the user process is not waiting.

3.1. User Address Space versus Kernel Address Space

A device driver is a part of the kernel. The kernel uses a completely different virtual address space from the virtual address space that a user process uses. When a device driver function is invoked through a system call, the driver must often map data from the user virtual address space to the kernel's virtual address space (most oftyen in the case of some DMA devices). Functions and macros are provdied to allow this 'dual' mapping of data. Normally the kernel can only access data that is addressable in its own address space.

3.2. User Context and Interrupt Context

A device driver has a *top half* and a *bottom half*. The top half is the part of the driver that runs only in the context of a user process making requests on the driver. The top half of a driver can start tasks which can cause long delays during which the system would want to switch to another process and continue doing useful work. When this happens the driver uses the *sleep* primitive to wait for a particular event to occur. Thus if a user program issues a *read* on (say) an A/D converter, the process would normally *sleep* until some input arrived. The driver could also use the *iowait* call for transfers that have already started.

The *bottom half* of a device driver is the part that runs at interrupt level. Thus in an A/D converter driver, the converter might interrupt when a sample was available. The bottom half of the driver could then store the data in a buffer and *wakeup* any user process sleeping in the top half so that that process could retrieve the data. If there was no user process sleeping in the top half, the *wakeup* would do nothing, but the next process to *read* the A/D driver would find the data already there and would not have to *sleep*.

3.3. Device Interrupts

Each hardware device interrupts (that is, the device *should* interrupt) at some *priority level*, trapping from wherever the system is currently executing, into the bottom half of the device driver at that priority level. This means that the *top half* of the device driver can be interrupted at any time by the bottom half of the driver. The top half and the bottom half share data structures which they wish to keep consistent. An example of such a data structure might be a pointer to a current buffer and a character counter. The top half of the driver must protect itself so that data structures can be updated as atomic actions, that is, the bottom half must not be allowed to interrupt during the time that the top half is updating some shared data structure. The way this protection is done is to bracket the critical sections of code (that updates or examines shared data structures) with a subroutine call that raises the processor priority to a level where the bottom half cannot interrupt. Such a piece of code looks like:

```
s = splx(hardware_priority);
        critical section of code which cannot be interrupted
(void)splx(s);
```

Note here that we raised the processor priority level and then restored the processor priority level after the protected section of code. (Determining the correct *hardware_priority* will be discussed later.) One section of code that almost always needs to be protected is the section where the top half checks to see if there is any data ready for it to read, or whether it can write data or start the device. Since the device can interrupt at any time, the section of code that checks for input in this fashion is wrong:

```
if (no input ready)
        sleep (awaiting input, software_priority)
```

because the device might well interrupt while the **if** condition is being tested, or while the preamble code for the *sleep* function is being executed.

The above section of code must be rewritten to look like this:

```
s = splx(hardware_priority);
while (no input ready)
        sleep (awaiting input, software_priority)
(void)splx(s);
```

If the top half executes the *sleep* system call, the bottom half will be allowed to interrupt, because the hardware priority level is reset to 0 as soon as the *sleep* context switches away from this process.

3.4. Interrupt Levels

In many cases it is possible to set the interrupt level a device will interrupt at by setting switches on the board. If so, you must decide what level this device is going to interrupt at. At first it may seem that your device is very high priority, but you must consider the consequences of locking out other devices:

- If you lock out the clock (level 5) time will not be accurate, and the UNIX scheduler will be suspended.
- If you lock out the on-board UARTS (level 6) characters may be lost.
- If you lock out the Ethernet (level 3), packets may be lost and retransmissions needed.
- If you lock out the disks (level 2), disk rotations may be missed.
- Level 1 is used for software interrupts and cannot be used for real devices.

In general, it is best to use level 2 to avoid the consequences of locking out other important system activities.

3.5. Some Common Service Functions

The kernel provides clusters of common service functions which device drivers can take advantage of. The common service functions fall into these major catagories:

Timeout Facilities
: are available when a device driver needs to know about real-time intervals.

Sleep and Wakeup Facilities
: suspend and resume execution of a process.

Raising and Lowering Interrupt Priorities
: Lock out devices by raising processor priority leve to stop the devices interrupting during critical operations (such as accessing shared data structures).

Multibus Resource Management
: includes the routines *mbsetup* and *mbrelse* for scheduling the Multibus resources.

Buffer Header Management
: Manages the in-memory disk buffer cache. We aren't dealing with disk drivers here so this needn't concern us.

There is also a kernel-specific version of the *printf* routine. The kernel *printf*is

3.5.1. Timeout Mechanisms

If a device needs to know about real-time intervals, *timeout(func, arg, interval)* is useful. *Timeout* arranges that after *interval* clock-ticks (fiftieths of a second) , the *func* is called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read a device if there is no response within a specified number of seconds (that is, there was a lost interrupt). Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general (you can't call *sleep* from within *func* for instance).

3.5.2. Sleep and Wakeup Mechanism

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, software_priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call returns when there is no process with higher *software_priority*.

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver (for a specific device if there is more than one minor device), which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened when they are awakened; they should check that the conditions which caused them to sleep no longer hold.

Software priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. In general, sleeps at less than PZERO should only be waiting for fast events like disk and tape i/o completion. Waiting for human activities like typing characters should be done at priorities greater than PZERO. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area '*u.*' should be touched, let alone changed, by an interrupt routine.

3.5.3. Raising and Lowering Interrupt Priorities

At certain places in a device driver it is necessary to raise the hardware interrupt priority so that a section of critical code cannot be interrupted, for example, while adding or removing entries from a queue, or modifying a data structure common to both halves of a driver.

The *splx* function changes the interrupt priority to a specified level, and returns a value which is what the level was before it changed.

For configuration reasons, the routine:

```
pritospl(mc->mc_intpri)
```

must be used to convert from the Multibus hardware interrupt level to the CPU hardware priority level. Here is how you normally use the *pritospl* and *splx* functions in a hypothetical *strategy* routine:

```
hypo_strategy(bp)
   register struct buf   *bp;
{
   register struct mb_ctlr    *mc = hypoinfo[minor(bp->b_dev)];
   int s;

   s = splx(pritospl(mc->mc_intpri));
   while (bp->b_flags & B_BUSY)
      sleep((caddr_t)bp, PRIBIO);
   . . .
      here is some critical code section
   . . .
   splx(s);     /* Set priority to what it was previously */
   . . .
}
```

3.5.4. Multibus Resource Management Routines

The routine *mbsetup* is called when the device driver wants to start up a transfer to the device using Multibus resource management.

At some later time, when the transfer is complete, the device driver calls the *mbrelse* routine to inform the Multibus resource manager that the transfer is complete and the resources are no longer required.

3.6. Kernel printf Function

The kernel provides a *prinf* function analogous to the *printf* function supplied with the standard I/O package for user programs. The kernel *printf* writes directly to the console however. When using the kernel *printf*, you should not use any floating-point conversions. The kernel *printf* function can be used to debug a driver.

3.6.1. Macros to Manipulate Device Numbers

A device number (in this system) is a 16-bit number divided into two parts called the *major* device number and the *minor* device number. There are macros provided for the purpose of isolating the major and minor numbers from the whole device number. The macro

```
major(dev)
```

returns the major portion of the device number *dev*, and the macro

```
minor(dev)
```

returns the minor portion of the device number. Finally, given a major and a minor number *x* and *y*, the macro

```
makedev(x,y)
```

creates a device number from the two portions.

3.7. Overall Layout of a Device Driver

Here is a summary of the kit of parts that comprises a typical device driver. In any given driver, some routines may be missing. In a complex driver, all of these routines may well be present. A typical device driver consists of a number of major sections, containing the routines described below.

Auto Configuration
: called by the kernel at system startup time to determine if the devices actually exist. This section contains the *probe* routine.

Opening and Closing the Device
: The *open* routine is called for each instance of an *open* or *create* request against that file. The *close* routine is called when a *close* request is made against that file for the last time.

Reading and Writing from or to the Device
: The *read* and *write* routines are called to get data from the device, or to send data to the device. The *read* and *write* routines may use the *tty* interfaces for devices such as terminals, or they might use a *strategy* routine to handle devices that transfer data in chunks. *Strategy* is most often used for DMA (Direct Memory Access) transfers, where the actual data buffer must be mapped in for the duration of the transfer.

Start Routine
: The *start* routine is called to actually initiate the I/O operation. *Start* is needed in drivers that queue requests; it is called from the *read*, *write* or *strategy* routine to start the queue and is also called from the interrupt routine to start the next element on the queue.

Mmap Routine
: The *mmap* routine is present in cases where it is required to map the device into user

memory — a frame buffer for instance.

Interrupt Routine

The interrupt routine of a device driver is called to service interrupts, possibly from the device for which this driver exists. However, there can be more than one device sharing the same interrupt level, and it is then also the task of the interrupt routine to determine if the interrupt is actually destined for this driver, or for some other driver.

Ioctl Routine

The *ioctl* routine is called when the user process does an *ioctl* system call. A typical use is to change the baud-rate for a serial interface.

3.8. A Very Basic Skeleton Device Driver

At this stage, we quit discussing the I/O system and start writing a very simple device driver. This model will be one of the simplest drivers we can produce. There is a complete version of this driver in the attachments to this manual — the parts are presented piecemeal here with some discussion on their functions.

What we do here is to invent an interface board called a Skeleton controller. The Skeleton board is a very simple I/O mapped board, that is, it uses I/O ports in the Multibus I/O address space. The Skeleton board has a single-byte command/status register, and a single-byte data register. You can only write data to the outside world from the Skeleton board. This board is not a slow teletype style interface — you can provide vast blocks of data and the board sends it all out very fast. The Skeleton board interrupts when it is ready for a data transfer. The board comes up in the power on state with interrupts disabled and everything else in a 'normal' state.

The status register of the Skeleton interface is located at 0x600 in Multibus I/O space, and the data register at 0x601. The status register is both a read and a write register. The bit assignments are as shown in the tables below.

BIT	8	7	6	5	4	3	2	1
Read	Inter-rupt				Device Ready	Interface Ready		Interrupt Enabled

BIT	8	7	6	5	4	3	2	1
Write						Reset		Enable Interrupt

Here is a brief description of what the bits mean:

When *reading* from the status register

bit 8 is a 1 when the board is interrupting, 0 otherwise.

bit 4 is a 1 when the device that the board controls is ready for data transfers.

bit 3 is a 1 when the Skeleton board itself is ready for data transfers.

bit 1 is a 1 when interrupts are enabled, 0 when interrupts are disabled.

When *writing* to the status register

bit 3 resets the Skeleton board to its startup state — interrupts are disabled and the board should indicate that it is ready for data transfers.

bit 1 enables interrupts by writing a 1 to this bit, disables interrupts by writing a 0.

The header file for this interface is in *skreg.h*. By convention, we put the register and control information for a given device (say *zy*) in a file called *zyreg.h*. The actual C code for the *zy* driver would by convention be placed in a file called *zy.c*. The header file for the Skeleton board looks like this:

```
/*
 * Registers for Skeleton Multibus I/O Interface
 */
struct sk_reg {
        char sk_data;   /* 01: Data Register */
        char sk_csr;    /* 00: command(w) and status(r) */
};

/* sk_csr bits (read) */
#define     SK_INTR  0x80 /* 1 if device is interrupting */
#define     SK_DEVREADY     0x08 /* Device is Ready */
#define     SK_INTREADY 0x04 /* Interface is Ready */
#define     SK_INTENAB   0x01 /* Interrupts are enabled */

/* sk_csr bits (write) */
#define     SK_RESET       0x04 /* reset the device and interface */
#define     SK_ENABLE      0x01 /* Enable interrupts */
```

The complete device driver for the Skeleton board consists of the following parts:

skprobe
: is the autoconfiguration routine called at system startup time to determine if the *sk* board is actually in the system.

skopen and *skclose*
: routines for opening the device for each time the file corresponding to that device is opened, and for closing down after the last file has been closed.

skwrite
: routine which is called to send data to the device.

skstrategy
: routine which is called from the *write* routine via *physio* to initiate transfers of data.

skstart
: routine which is called for every byte to be transferred.

skintr
: the interrupt routine which services interrupts and arranges to transfer the next byte of data to the device.

The subsections to follow describe these routines in more detail.

3.9. General Declarations in Driver

In addition to including a bunch of system header files, there are some data structures which the driver must define.

```
#include "sk.h"        /* header file generated by config (defines NSK) */

#define   SKPRI    (PZERO-1)/* software sleep priority for sk */

#define   SKUNIT(dev)   (minor(dev))

struct    buf rskbuf[NSK];

int skprobe(), skintr();

u_long skaddrs[] = { 0x600, 0};

struct    mb_device *skdinfo[NSK];
struct mb_driver skdriver = { skprobe, 0, 0, 0, 0, skintr,
                 skaddrs, 0, 0, 0, 0, "sk", skdinfo, 0,
};

struct sk_device {
     struct buf *sk_bp;    /* current buf */
     int   sk_count;  /* number of bytes to send */
     char *sk_cp;     /* next byte to send */
     char sb_busy;    /* true if device is busy */
} skdevice[NSK];
```

Here's a brief discussion on the declarations in the above example.

sk.h	file is generated by the *config* program (discussed later). It contains the definition of NSK, the number of sk devices configured into the system.
SKPRI	declaration declares the software priority level at which this device driver will sleep.
SKUNIT	macro is a common way of obtaining the minor device number in a driver. Study just about any device driver and you will find a declaration like this — it is a stylized way of referring to the minor device number. One reason for this is that sometimes a driver will encode the bits of the minor device number to mean things other than just the device number, so using the SKUNIT convention is an easy way to make sure that is things change, the code will not be affected.
rskbuf	array is necessary so that there will be *buf* structures to pass to the *physio* routine. *Physio* will fill in certain fields before calling our *strategy* routine with the *buf* structure as the argument.

• Then there is a definition of the system dependent entry points into the device driver. In this driver, the only entry points we use are *skprobe* (probes the Multibus during system configuration time) and *skintr* (interrupt routine).

skaddrs	is the list of addresses that this device appears on in the Multibus address space. The address of this array appears in the *skdriver* strucure defined below.

skdinfo	is the *device* structure for this driver. The system autoconfiguration routines fill in the apporpriate fields in this structure at startup time.
skdriver	is a definition of the *driver* structure for this driver. An explanation of the fields in this structure and when they should be filled in appears earlier in this chapter.
sk_device	is a definition of a structure that holds state information for each unit. This is information specific to this driver that needs to be remembered between subroutine calls.

3.10. Autoconfiguration Procedures

Part of a device driver's work is handling the automatic determination of the system configuration. When the Sun UNIX system boots up, it determines the peripheral configuration details by probing the Multibus memory space and Multibus I/O space of the machine.

Note that the autoconfiguration routines make some assumptions about where things are in the system:

- Any address less than 64K is assumed to be Multibus I/O address space.
- Addresses less than 256K are assumed to be for DVMA purposes.
- The autoconfiguration routines search the addresses specified in the configuration file as well as the addresses specified in the driver.

3.11. Probe Routine

There should be a *probe* function in every driver. *Probe* is called at system initialization time with an address to be probed. *Probe* has two functions:

1. To determine if the device that this driver is written for exists at the specified address, and:
2. To make the kernel aware of how much of the system's resources to reserve for that device.

Under normal circumstances, addressing non-existent memory or I/O space on the Multibus generates a bus error in the CPU. The kernel provides some functions to probe the address space, recover from possible bus errors, and return an indication as to whether the attempt to address a specific location generated a bus error.

Determining whether a device actually exists or not is assisted by the functions *peek*, *peekc*, *poke*, and *pokec*. These functions provide for accessing possibly non-existent addresses on the bus without generating bus errors that would terminate the process trying to access those addresses. *Peek* and *poke* read and write, respectively, 16-bit words (short's in the Sun system). *Peekc* and *pokec* read and write 8-bit characters. In general, you will use the character routines for probing single-byte I/O registers. See the section *Summary of Functions* for details on these routines.

Having determined whether the device exists in the system, the *probe* function returns either:

- the size (in bytes) of the device structure if it does exist. The kernel uses the value returned from *probe* to reserve memory resources for that device. For I/O mapped devices, *probe* returns the amount of Multibus I/O space that the device registers consume. For memory-mapped devices, *probe* returns the amount of memory that the device consumes.
- a value of 0 (zero) if the device does not exist.

Now we can write *skprobe*:

```
skprobe(reg, unit)
        caddr_t reg;
        int unit;
{
        register struct sk_reg *sk_reg;
        register int c;

        sk_reg = (struct sk_reg *)reg;
        c = peekc((char *)&sk_reg->sk_csr);
        if (c == -1)
                return (0);

        return (sizeof (struct sk_reg));
}
```

The *reg* argument is the purported address of the device. The *unit* argument is usually ignored.

If the *probe* routine determines that the device actually exists and it returns the amount of resources that the deevice uses, the system startup routines set the *md_alive* field in the device structure to non-zero. The *md_alive* field is then used subsequently by other driver functions to check that the device was probed successfully at startup time.

3.12. Open and Close Routines

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device driver to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

The *Open* routine for the **sk** driver is simple. *Skopen* is called with two arguments, namely, the device which must be opened, and a flag indicating whether the device should be opened for reading, writing, or both. The first task is to check whether the device number to be opened actually exists — *skopen* returns an error indication if not. The second check is whether the open is for writing. Since **sk** is a 'write only' device, it is an error to open it for reading only. If all the checks succeed, *skopen* enables interrupts from the device, and then returns a zero (0) as an indication of success. Here is the code for the *skopen* routine:

```
skopen(dev, flags)
	dev_t dev;
	int flags;
{
	register struct mb_device *md;
	register struct sk_reg *sk_reg;

	if (SKUNIT(dev) >= NSK ||
	   (md = skdinfo[SKUNIT(dev)]) == 0 || md->md_alive == 0)
		return (ENXIO);

	if (flags & FREAD)
		return (ENODEV);

					/* enable interrupts */
	sk_reg = (struct sk_reg *)md->md_addr;
	sk_reg->sk_csr = SK_ENABLE;

	return (0);
}
```

The first if statement checks if the device actually exists. Note the use of the SKUNIT macro to obtain the minor device number — we discussed this earlier on.

The *close* routine for the **sk** driver is very simple — all it does is disable interrupts:

```
/*ARGSUSED*/
skclose(dev, flags)
	dev_t dev;
	int flags;
{
	register struct mb_device *md;
	register struct sk_reg *sk_reg;
	md = skdinfo[SKUNIT(dev)];

					/* disable interrupts */
	sk_reg = (struct sk_reg *)md->md_addr;
	sk_reg->sk_csr |= ~SK_ENABLE;
}
```

Skclose could in fact be more complicated than this. Some of the actions that could take place in a *close* routine might be to deallocate any resources that were allocated for this device driver, and possibly to *sleep* on completion of I/O transfers for that device.

3.13. Read and Write Routines

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *iovec.iov_base*, *iovec.iov_len*, and *uio.uio_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate *read* or *write* routine is called — this *read* or *write* routine is responsible for transferring data and updating the count and current location appropriately as discussed below.

The *write* routine for the skeleton driver is very simple. *Write* simply calls the *strategy* routine through the *physio* system routine. *Physio* ensures that the user's memory space is available to the driver for the duration of the data transfer. *Physio* also takes care of updating the count and current location as appropriate. The *write* routine looks like this:

```
skwrite(dev, uio)
      dev_t dev;
      struct uio *uio;          see below for some notes on this
{

      if (SKUNIT(dev) >= NSK)
            return (ENXIO);
      return (physio(skstrategy, &rskbuf[SKUNIT(dev)], dev, B_WRITE,
                skminphys, uio));
}
```

The *skminphys* routine is called by *physio* to determine the largest reasonable blocksize to transfer at once. If the user has requested more bytes than this, *physio* will call *skstrategy* repeatedly, requesting no more than this blocksize each time. The case where this is important is when DVMA transfers are done. (DVMA is covered in more detail below.) The reasoning is that only a finite amount of address space is available for DMVA transfers and it is not reasonable for any device to tie up too much of it. A disk or a tape might reasonably ask for as much as 64 Kbytes; slow devices like printers should only ask for one to four Kbytes since they will tie up the resource for a relatively long time.

Here is the *skminphys* routine.

```
skminphys(bp)
      struct buf *bp;
{
      if (bp->b_bcount > MAX_SK_BSIZE)
            bp->b_count = MAX_SK_BSIZE;
}
```

Note that if you don't suppy you own *minphys* routine, you can simply place a zero (0) as the argument to the *strategy* routine at that place, and the system supplied *minphy* routine gets used instead.

3.13.1. Some Notes About the UIO Structure

When the system is reading and writing data from or to a device, the *uio* structure is used extensively. The *uio* structure is a general structure to allow for what is called gather-write and scatter-read. That is, when writing to a device, the blocks of data to be written don't have to contiguous in the user's memory but can be in physically discontiguous areas. Similarly, when reading from a device into memory, the data comes off the device in a continuous stream but can go into physically discontiguous reas of the user's memory. Each discontiguous area of memory is described by a structure called an *iovec* (I/O vector). Each *iovec* contains a pointer to the data area to be transferred, and a count of the number of bytes in that area. The *uio* structure describes the complete data transfer. *Uio* contains a pointer to an array of these *iovec* structures. Thus when you want to write a number of physically discontiguous blocks of memory to a device, you can set up an array of *iovec* structures, and place a pointer to the start of the array in the *uio* structure. In the trivial case, there is generally just one block of data to be transferred, and so the *uio* structure is fairly simple.

3.14. Skeleton Strategy Routine

The *strategy* routine is called by *physio* after the user buffer has been locked into memory. The *strategy* routine must check that the device is ready and initiate the data transfer. *Strategy* will then wait for the the completion of the data transfer, which will be signaled by the interrupt routine.

```
skstrategy(bp)
        register struct buf *bp;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int s;

        md = skdinfo[SKUNIT(bp->b_dev)]
        sk_reg = (struct sk_reg *)md->md_addr;
        sk = &sk_device[SKUNIT(dev)];
        s = splx(pritospl(md->md_intpri));
        while (sk->sk_busy)
                sleep((caddr_t) sk, SKPRI);
        sk->sk_busy = 1;
        sk->sk_bp = bp;
        sk->sk_cp = bp->b_un.b_addr;
        sk->sk_count = bp->b_bcount;
        skstart(sk, (struct sk_reg *)md->md_addr,);
        sk->sk_busy = 0;
        wakeup((caddr_t) sk);
        splx(s);
}
```

3.15. Skeleton Start Routine — Initiate Data Transfers

The *start* routine is responsible for getting the actual data bytes out to the device itself. *Start* is called once by *strategy* to get the very first byte out to the interface. After that, it is assumed that the device will interrupt every time it is ready for a new data byte, and so *start* is thereafter called from the interrupt routine. Here is the *start* routine:

```
skstart(sk, sk_reg)
        struct sk_device *sk;
        struct sk_reg *sk_reg;
{
        sk_reg->sk_data = *sk->sk_cp++;
        sk->sk_count--;
        sk_reg->sk_csr = SK_ENABLE;
}
```

This routine will work, but there is a lot of overhead in taking an interrupt from the device on every character. Since we know that the device can take characters very quickly. it would be more efficient to try to give characters quickly. What we will do is to check after each character and give another one if the device is ready. Here is the new, more efficient *skstart* routine.

```
skstart(sk, sk_reg)
        struct sk_device *sk;
        struct sk_reg *sk_reg;
{
        do {
                sk_reg->sk_data = *sk->sk_cp++;
                sk->sk_count--;
        } while (sk->sk_count && sk_reg->sk_csr & SK_DEVREADY);
        if (sk->sk_count)       /* more characters to go */
                sk_reg->sk_csr = SK_ENABLE;
        else {
                sk_reg->sk_csr = 0;   /* disable interrupts */
                iodone(sk->sk_bp);
        }
}
```

We give characters to the device as long as there are more characters and the device is ready to receive them. If we run out of characters, we disable interrupts to keep the device from bothering us and call *iodone* to mark the buffer as done.

It may be that the device is not quite quick enough to take a character and raise the *SK_DEVREADY* bit in the time we can decrement and test the counter. If so, it would be very worthwhile to busy wait for a short time. The reasoning is that while busy waiting is a waste, servicing an interrupt costs lots more CPU time, and if busy waiting works fairly often it is a big win. There is a macro *DELAY* which takes an integer argument which is approximately the number of microseconds to delay, so we could add

```
        DELAY(10);
```

just before the while. Clearly this is an area where experimentation with the real device is called for.

3.16. Interrupt Routines

Each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. After the interrupt has been processed, a return from the interrupt handler returns from the interrupt itself.

The address of the interrupt routine for a particular device driver is contained in the per-driver (that is, *mb_driver*) data structure for that device driver. The address of the interrupt routine is filled in statically at the time the data structure is declared and initialized.

Since there may be many devices sharing a common interrupt level, it is the specific driver's responsibility to determine if the interrupt is intended for it or not. If the interrupt *is* for this driver, the driver must service the interrupt and return a non-zero value to indicate that the interrupt has been serviced. If the interrupt is *not* for this device driver, the interrupt routine must return a zero value.

It is expected that the device actually indicates when it is interrupting. If there are any more bytes to transfer, the interrupt routine calls the *start* routine to transfer the next byte. If there are no more bytes to transfer, the interrupt routine disables the interrupt (so that the device won't keep interrupting when there is nothing to do), and finishes up by calling *iodone*.

```
skintr()
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int serviced;

        serviced = 0;
        for (i = 0; i < NSK; i++) {
                md = &skdinfo[i];
                sk_reg = (struct sk_reg *)md->md_addr;
                if (sk_reg->sk_csr & SK_INTR) {
                        serviced = 1;
                        sk = &sk_device[i];
                        if (sk->sk_count == 0) {
                                sk_reg->sk_csr = 0;     /* disable interrupts */
                                iodone(sk->sk_bp);
                        } else
                                skstart(sk, sk_reg);
                }
        }
        return (serviced);
}
```

3.17. Ioctl Routine

The *ioctl* routine is used to perform any tasks that can't be done by the regular *open*, *close*, *read*, or *write* routines. Typical applications are: 'what is the status of this device', or 'tell me the partitions on disk **xy1**'. This device does not need any special functions so we don't have an *ioctl* routine.

3.18. Devices That Do DMA

Devices that are capable of doing DMA are treated a little differently than the skeleton device we have been working with so far. Let us assume that we have a new version of the skeleton board; call it the Skeleton II. It can do DMA transfers and we want to use this feature since it is much more efficient. First we must describe DMA on the Sun-2.

3.19. Multibus DVMA

On the Sun-2, the processor board is always listening to the Multibus for memory references. When a request to read or write Multibus memory between addresses 0 and 256K comes up, the DVMA hardware takes the address, adds 0xF00000 to it, and goes through the kernel memory map to find the location in processor memory that will be used. Thus if you wish to do DMA over the Multibus, you must make the appropriate entries in the kernel memory map. As you might expect, there are subroutines to help with this chore. *Mbsetup* sets up the map and *mbrelse* releases the map.

3.20. Changes to the Driver

The changes to the driver are surprisingly simple. FIrst we must extend the *sk_reg* structure which defines the device registers. We assume that the Skeleton II supports the following structure.

```
struct sk_reg {
        char sk_data;     /* 01: Data Register */
        char sk_csr;      /* 00: command(w) and status(r) */
        short     sk_count; /* bytes to be transferred */
        caddr_t   sk_addr;  /* DMA address */
};
```

Next we assume another bit in the csr.

```
#define    SK_DMA  0x10 /* Do DMA transfer */
```

And we must add another element in the *sk_device* structure for use by *msetup* and *mbdone*.

```
        int  sk_mbinfo;
```

Now we change the *skstrategy* routine to use the DMA feature.

```
skstrategy(bp)
        register struct buf *bp;
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct struct sk_device *sk;
        int s;

        md = skdinfo[SKUNIT(bp->b_dev)]
        sk_reg = (struct sk_reg *)md->md_addr;
        sk = &sk_device[SKUNIT(dev)];
        s = splx(pritospl(md->md_intpri));
        while (sk->sk_busy)
                sleep((caddr_t) sk, SKPRI);
        sk->sk_busy = 1;
        sk->sk_bp = bp;
        /* this is the part that is changed */
        sk->sk_mbinfo = mbsetup(md->md_hd, bp, 0);
        sk_reg->sk_count = bp->b_count;
        sk_reg->sk_addr = MBI_ADDR(sc->sc_mbinfo);
        sk_reg->sk_csr = SK_ENABLE | SK_DMA;
        /* end of changes */
        iowait(bp);
        sk->sk_busy = 0;
        wakeup((caddr_t) sk);
        splx(s);
}
```

The need for the *skstart* routine is completely gone and thus we will delete it. All the i/o now is started by *skstrategy* and continues until *skintr* is called. Thus we can delete the *sk_cp* and *sc_count* variables from the *sk_device* structure.

Skintr is also simplified. There is no longer any need to check the count since all the data goes out through DMA. Therefore *iodone* will always be called. Also, we need to free up the Multibus resources, so we will call the *mbrelse* routine. Here is the new *skintr* routine:

```
skintr()
{
        register struct mb_device *md;
        register struct sk_reg *sk_reg;
        register struct sk_device *sk;
        int serviced;

        serviced = 0;
        for (i = 0; i < NSK; i++) {
                md = &skdinfo[i];
                sk_reg = (struct sk_reg *)md->md_addr;
                if (sk_reg->sk_csr & SK_INTR) {
                        serviced = 1;
                        sk = &sk_device[i];
                        /* this is the part that is changed */
                        sk_reg->sk_csr = 0;     /* disable interrupts */
                        mbrelse(md->md_hd, &sk->sk_mbinfo);
                        iodone(sk->sk_bp);
                        /* end of changes */
                }
        }
        return (serviced);
}
```

3.21. Errors

We have been pretty casual about errors up till now. Most devices have at least an error bit in the csr, and usually more detailed error information is available. Also, we should check whether the DMA count is exhausted.

Detection and treatment of errors varies greatly from device to device and is not very generalisable, so it wouldn't add much to this tutorial to show some elaborate error checking. Nonetheless, error checking is important because if you don't check for errors and they do happen your users will be very unhappy.

You should read the Product Specification manual for your device very carefully to determine what error indications can be given and what you should do when they do come up. At the very least, check for errors and if you can't figure out what to do about them, printf a message to the console just to let the world know that everything is not perfectly OK.

3.22. Memory Mapped Devices

Devices such as frame buffers are frequently accessed by mapping the buffer into the user address space and allowing the user to update them at will. The user accomplishes this through a *mmap*(2) system call. This call is translated by the kernel into a call to the driver's *mmap* routine. The call has three parameters, *dev, off* and *prot*. *Dev* is of course the device major and minor number, *off* is the offset into the frame buffer from the user's *mmap* system call, and *prot* is a flag indicating whether write protection applies to the page(s). The constants *PROT_READ*, *PROT_WRITE* and *PROT_EXEC* are defined in the header file *mman.h*. Each constant is a bit turned on to indicate that the appropriate access is allowed.

Here is the *mmap* routine from the Sun Color Graphics driver.

```
cgmmap(dev, off, prot)
      dev_t dev;
      off_t off;
      int prot;
{
      register caddr_t addr;
      register int page, uc;

      addr = cginfo[minor(dev)]->md_addr;
      if (off >= CGSIZE)
            return (-1);
      page = getkpgmap(addr + off) & PG_PFNUM;
      return (page);
}
```

The *PG_PFNUM* constant gets rid of extraneous bits that *getkpgmap* returns and just leaves the page number, which is what we have to return.

The routine first gets the address of the frame buffer from the Multibus device structure. Remember that this is generated by *config* based on the user's input as to where devices are configured. Next the offset is checked to be sure the user isn't mapping beyond the end of the frame buffer. Next comes a call to *getkpgmap* to do the actual mapping. The page number returned by *getkpgmap* is then returned by *cgmap*. In this case, *prot* is not checked since the driver permits open to succeed only if the user is opening for both read and write, thus all access are permitted.

4. Configuring the System to Add Skeleton Driver

Now we've written the Skeleton driver, we'll go through the steps required to add it to the system. A detailed description of how to configure and build a kernel is in the document *Building UNIX Systems With Config* in the *System Manager's Manual*. Here we just cover what is needed to add a new driver.

New device drivers require entries in */sys/sun/conf.c* and in */sys/conf/files.sun*. They are included by mentioning the device name in the configuration file.

The examples to follow assume that you are adding a driver for the Skeleton board (sk) to the system. The new system will be called *SKELETON*. Here is a representative section from *sun/conf.c*:

```
#include "sk.h"
#if NSK > 0
int     skopen(), skclose(), skread(), skwrite(), skmmap();
#else
#define  skopen    nodev
#define  skclose   nodev
#define  skread    nodev
#define  skwrite   nodev
#define  skmap     nodev
#endif
   {
skopen,   skclose,   skread,   skwrite,      /* 30 */
nodev,         nodev,     nodev,    0,
seltrue,       skmmap,
   },
```

If *NSK* is greater than 0, this will add the driver routines into the *cdevsw* table so the kernel knows where they are. (*NSK* is set by the *config* program based on the kernel configuration file discussed below.) The entres added are, in order, the *open*, *close*, *read*, *write*, *ioctl*, *stop* and *reset* routines, a *tty* structure address and finally the *select* and *mmap* routines. We do not have an *ioctl* routine so this entry calls *nodev* which is a special routine that always returns an error. Since we are not a tty we do not have a *stop* routine which would be used for flow control, nor do we have a *tty* structure. The *reset* routine is not used so all devices use *nodev* for this one. The *select* routine is called when a user process does a *select*(2) system call; it returns true if the device can be immediately selected. Since our sk device is write only and fast, it is always selectable so we use the default *seltrue* routine which always returns true.

Here is the line you must add to *files.sun*:

```
sundev/sk.c       optional sk device-driver
```

This says that the file *sundev/sk.c* contains the source code for the optional *sk* device and that it is a device driver.

Now, you can go through the process of building the system just as described in the chapter on configuration: Choose a name for your configuration of the system — in our case it will be called *SKELETON*. Then create the configuration file and directory:

```
gaia#  cp GENERIC SKELETON
gaia#  mkdir ../SKELETON
```

Edit *SKELETON* to reflect your system — you must add a description of the device to the *SKELETON* file:

```
        device      sk0 at mb0 csr 0x600 priority 3
```

This entry says we have an sk device (the first device is always number 0) on the Multibus, the control/status register (device register) is at Multibus address 0x600 (this is passed to out *probe* routine at boot time), and that this deivce will interrupt at level 3.

Then you can run */usr/etc/config* to make the configuration files for the new device driver:

```
        gaia# /usr/etc/config SKELETON
```

/usr/etc/config uses *SKELETON*, *files*, and *files.sun* as input, and generates a number of files in the ../*SKELETON* directory.

Now you can change directory to the new configuration directory, ../*SKELETON* in this case, and make the new system:

```
        gaia# cd ../SKELETON
        gaia# make depend
        gaia# make
```

The **make depend** command creates the dependency tree for any new C source files you might have created during the process of adding new drivers or whatever to the system.

Now you must add a new device entry to the */dev* directory. The connections between the UNIX operating system kernel and the device driver is established through the entries in the */dev* directory. Using the example above as our model, we want to install the device for the Skeleton driver.

Making new device entries is done via a shell script called *MAKEDEV* in the */dev* directory. It is worth while looking inside *MAKEDEV* to find out the kinds of things that go on in there. The lines of shell script below reflect what you would add to *MAKEDEV* for the new Skeleton device. First, there are lines of commentary at the start of the *MAKEDEV* file:

```
        #! /bin/sh
        #    MAKEDEV      4.3    83/03/31
        # Graphics
        #    sk*   Skeleton Board
```

Then there is the actual shell 'code' which makes the device entries:

```
        skeleton|sk|sk0)
                /etc/mknod sk0 c 30 0              ; chmod 666 sk0
                ;;
```

This makes the special inode **/dev/sk0** as a character special device with major device number 30 and minor device number 0, and then sets the mode of the file so that anyone can read or write the device.

Having added the new device entry, you can install the new system and try it out.

```
gaia# cp vmunix /vmunix+
gaia# /etc/halt
        The system goes through the halt sequence, then
        the monitor displays its prompt, at which point you
        can boot the system in single-user state
> b vmunix+ -s
        The system boots up in single user state and
        then you can try things out
gaia#
```

If the system appears to work, save the old kernel under a different name and install the new one in /vmunix:

```
gaia# cd /
gaia# mv vmunix ovmunix
gaia# mv vmunix+ vmunix
gaia#
```

Make sure that the new version of the kernel is actually called *vmunix* — because programs such as *ps* and *netstat* use that exact name to look for things, and if the running version of the kernel is called something other than *vmunix* the results from such programs will be wrong.

5. Summary of Functions

5.1. Standard Error Numbers

The system has a collection of standard error numbers that a driver can return to its callers. These numbers are described in detail in *intro*(2), the introductory pages of the *System Interface Manual*. A complete listing of the error numbers appears in *<sys/errno.h>*.

5.2. Device Driver Routines

5.2.1. Autoconfiguration Routines

5.2.1.1. Probe — Determine if Hardware is There

```
probe(reg)
    caddr_t reg;
```

Probe determines whether the device at address *reg* actually exists and is the correct device for this driver. If the device exists and is correct, *probe* returns

```
return (sizeof (struct device));
```

If the device does not exist, or is the wrong device for this driver, *probe* returns 0 (zero).

5.2.2. Open and Close Routines

5.2.2.1. Open — Open a Device for Data Transfers

```
open(dev, flags)
    dev_t dev;
    int flags;
```

Open checks that the minor device number passed in the *dev* argument is in range. The integer argument *flags* contains bits telling whether the open is for reading, writing, or both. The constants *FREAD* and *FWRITE* are available to be and'ed with *flags*. *Open* returns:

```
return (ENXIO);
```

(meaning a non-existent device) if the minor device number is out of range. Then *open* attempts to initialize the device, and if there are any errors, *open* returns:

```
return (EIO);
```

to mean an I/O error. If the open is successful, *open* returns 0 (zero).

5.2.2.2. Close — Close a Device

```
close(dev)
    dev_t dev;
    int flags;
```

Close does whatever it has to do to indicate that data transfers cannot be made on this device until it has been reopened. *Flags* is the same as for *open*.

5.2.3. Read, Write, and Strategy Routines

5.2.3.1. Read — Read Data from Device

```
read(dev, uio)
    dev_t dev;
    struct uio *uio;
```

Read is the high-level routine called to perform data transfers from the device. *Read* must check that the minor device number passed to it is in range. If the minor device number is out of range, *read* returns:

```
return (ENXIO);
```

meaning that the device is non-existent. Subsequent actions of *read* differ depending on whether the device is a character-at-a-time device such as a teletype, or is a block transfer device.

For the block-transfer devices, *read* simply calls on the *strategy* function via *physio*:

```
return (physio(strategy, &rbuf[minor(dev)], dev, B_READ, minphys, uio));
```

5.2.3.2. Write — Write Data to Device

```
write(dev, uio)
    dev_t dev;
    struct uio *uio;
```

Write is the high-level routine called to perform data transfers to the device. *Write* must check that the minor device number passed to it is in range. If the minor device number is out of range, *write* returns:

```
    if (VPUNIT(dev) >= NVP)
        return (ENXIO);
```

Subsequent actions of *write* differ depending on whether the device is a character-at-a-time device such as a teletype, or is a block transfer device.

For the block-transfer devices, *write* simply calls on the *strategy* function via *physio*:

```
        return (physio(strategy, &rbuf[minor(dev)], dev, B_WRITE, minphys, uio));
```

5.2.3.3. Strategy Routine

```
        strategy(bp)
            register struct buf *bp;
```

Strategy is the high level routine responsible for getting the data to the actual device. For DMA devices, *strategy* calls on *mbgo* to schedule the Multibus resources. *strategy* does not return any value.

5.2.3.4. Minphys — Determine Maximum Block Size

```
        int    block = some 'reasonable' block size for transfers
                       must be a multiple of 1024 bytes

        unsigned minphys(bp)
            register struct buf *bp;
```

Minphys determines a 'reasonable' block size for transfers, so as to avoid tying up too many resources. *Minphys* is passed as an argument to *physio*. In the absence of a *minphys* functions supplied by the device driver itself, a system supplied version of *minphys* is used instead. *Minphys* shoulld perform the calculation:

```
            if (bp->b_bcount > block)
                bp->b_bcount = block;
```

5.2.4. Ioctl — Special Interface Function

```
        ioctl(dev, cmd, data, flag)
            dev_t dev;
            int cmd;
            caddr_t data;
            int flag;
```

Ioctl differs for every device and covers the functions that aren't done by *read* and *write*. *Ioctl* does whatever it has to do, then returns 0 (zero) if there were no errors, and returns:

```
        return (ENOTTY);
```

in the case that the command requested did not apply to this device. Note that ENOTTY gives rise to the error message 'Not a typewriter', which may be misleading.

5.2.5. Low Level Routines

Routines in this area are low level and can potentially be called from the interrupt side of the driver. *Sleep* calls may never be made from the routines described here.

5.2.5.1. Intr — Handle Interrupts

```
intr()
```

Intr is responsible for fielding interrupts from the device. In situations where more than one device share the same interrupt level, *intr* must determine if the interrupt was actually detsined for this driver or not. *Intr* returns 0 (zero) to indicate that the interrupt was not serviced by this driver, and non-zero to indicate that the interrupt was serviced. It is a gross error for *intr* to say that it serviced an interrupt when it really did not.

5.3. Common Service Routines

5.3.1. Sleep — Sleep on an Event

```
sleep(address, priority)
   caddr_t  address;
   int priority;
```

Sleep is called to put a process to sleep. The *address* argument is typically the address of a location in memory. *Priority* is the software priority the process will have after it is woken up. The process which has been put to sleep can be woken up again by issuing a *wakeup* call with the same *address*. *Sleep* should *never* be called from the low level side of a driver.

5.3.2. Wakeup — Wake Up a Process Sleeping on an Event

```
wakeup(address)
   caddr_t  address;
```

Wakeup is called when a process waiting on an event must be awakened. *Address* is typically the address of a location in memory. *Wakeup* is typically called from the low level side of a driver when (for instance) all data has been transferred to or from the user's buffer and the process waiting for the transfer to complete must be awakened.

5.3.3. Mbsetup — Set Up to Use Multibus Resources

```
mbsetup(md_hd, bp, flag)
   struct mb_hd      *mb_hd;
   struct buf   *bp;
   int      flag;
```

Mbsetup is called to set up the memory map for a Multibus DVMA transfer. *flag* is *MB_CANTWAIT* if the caller desires not to wait for map resources if none are available. Normally this will be zero which means the driver will wait. *Mbsetup* returns an integer which must be saved for the call to *mbrelse*.

5.3.4. Mbrelse — Free Multibus Resources

```
mbrelse(md_hd, mbinfop)
   struct mb_hd      *mb_hd;
   int      *mbinfop;
```

Mbrelse releases the Multibus DVMA resources allocated by *mbsetup*. Note that the second parameter is a *pointer* to the integer returned by *mbsetup*.

5.3.5. Physio — Lock in User's Buffer Area

```
physio(strat, buf, dev, flag, minphys, uio)
   void  (*strat) ();
   struct  buf  *buf;
   dev_t  dev;
   int  flag;
   void  (*minphys) ();
   struct  uio  *uio;
```

5.3.6. Iowait — Wait for I/O to Complete

```
iowait(bp)
   struct  buf  *bp;
```

Iowait waits on the buffer header addressed by *bp* for the DONE flag to be set. *Iowait* actually does a *sleep* on the buffer header.

5.3.7. Iodone — Indicate I/O Complete

```
iodone(bp)
   struct  buf  *bp;
```

Iodone is called to indicate that I/O associated with the buffer header *bp* is complete. *Iodone* sets the DONE flag in the buffer header, then does a *wakeup* call with the buffer pointer as argument.

5.3.8. Pritospl — Convert Priority Level

```
pritospl(value)
   int value;
```

Pritospl converts the hardware priority level given by *value*, which is a Multibus priority level, to a CPU hardware priority level used by *splx*. *Pritospl* is used to parameterize the setting of priority levels.

5.3.9. spl*n*() — Set Specific Priority Level

The *spln*() functions are available for setting the priority level to *n*, where *n* ranges from 0 to 7. These routines should probably never be used in any device driver.

5.3.10. splx — Reset Priority Level

```
splx(s)
   int s;
```

Splx called with an argument *s* sets the priority level to *s*. *Splx* is typically used to restore the priority level to a previously remembered level.

5.3.11. uiomove — move data to or from the uio structure

```
uiomove(cp, n, rw, uio)
   register caddr_t cp;
   register int n;
   enum uio_rw rw;
   register struct *uio;
```

Device drivers use *uiomove* to move a specified number of bytes between an area defined by a *uio* structure (normally passed to the driver when it is called) and an area in the kernel's address space (where it can be used by the driver). *Uiomove* moves *n* bytes from or to the *iovec* pointed to by the *uio* structure out of or into the area specified by *cp*. The read/write flags (which specify the direction of the data transfer) are defined in <*uio.h*>. *Uiomove* replaces the older *copyin* and *copyout* routines which are no longer supported. *Uiomove* can also be used to copy kernel *uio* structures — it checks *uio->uio_segflag*.

5.3.12. ureadc and uwritec — transfer bytes to or from a uio structure

```
ureadc(c, uio)
   int c;
   register struct *uio;
```

Ureadc transfers a character represented by *c* in the definition into the *iovec* pointed at by the *uio* structure (normally passed to the driver when it is called). *Ureadc* is normally used when 'reading' a character in from a device.

```
char
uwritec(uio)
   register struct *uio;
```

Uwritec Ureadc returns the next character in the *iovec* pointed at by the *uio* structure (normally passed to the driver when it is called). *Uwritec* is normally used when 'writing' a character out to a device.

Note that 'read' and 'write' are slightly confusing in the above contexts, since *ureadc* actually obtains a character from somewhere and places the character *into* the *iovec* pointed to by the *uio* structure, whereas *uwritec* obtains a character from the *iovec* and 'writes' the character somewhere.

Ureadc and *uwritec* replace the routines *cpass* and *passc*, which are no longer supported.

5.3.13. peek, peekc — Check Whether an Address Exists and Read

```
peek(address)
   short *address;

peekc(address)
   char *address;
```

peek and *peekc* are called with an address from which you want to read. Both *peek* and *peekc* return -1 if the addressed location doesn't exist, otherwise they return the value which was fetched from that location.

5.3.14. poke, pokec — Check Whether an Address Exists and Write

```
poke(address, value)
   short *address;
   short value;

pokec(address, value)
   char *address;
   char value;
```

poke and *pokec* are called with an *address* you want to store into, and *value* is the value you want to store there. Both *poke* and *pokec* return 1 if the addressed location doesn't exist, and 0 if the addressed location does exist.

5.3.15. geteblk — Allocate Dynamic Buffer

```
struct buf *geteblk(size)
   int *size;
```

geteblk allocates a buffer dynamically. The *size* of the block is limited to a maximum of 8K bytes, and must be a multiple of 512 bytes.

5.3.16. brelse — Free Dynamic Buffer

```
brelse(bp)
struct buf bp;
```

brelse frees a buffer previously allocated by *geteblk*.

5.3.17. swab — Swap Bytes

```
swab(from, to, nbytes)
caddr_t from;
caddr_t to;
int nbytes;
```

swab swaps bytes within words. *nbytes* is the number of bytes to swap, and is rounded up to a multiple of two. The *from* and *to* areas can overlap each other since the bytes are swapped one at a time.

Appendix A. Sample Drivers

The C code listings suppiled here are sample drivers for devices that the Sun system supports. There are three drivers listed here:

CGONE
: is a device driver for the Sun-1 color graphics board. It is one of the simplest drivers around, being memory mapped.

SKY
: is a programmed I/O driver for the SKY floating-point board.

VP is a fairly good example of a DMA device driver.

```
/*      @(#)cgreg.h 1.2 83/08/16 SMI     */

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Register definitions for Sun Color Board
 */
#define CGSIZE  (16*1024)         /* 16K of address space */

# define GR_bd_sel   CGXBase      /* Select Color Board */

# define GR_x_select  0x0800      /* Access a column in the frame buffer */
# define GR_y_select  0x0000      /* Access a row in the frame buffer */
# define GR_y_fudge   0x0200      /* Bit 9 not used at all */
# define GR_update    0x2000      /* Update frame buffer if this bit set */
# define GR_x_rhaddr  0x1b80      /* Location to read X address bits A9-A8.
                                     Data put into D1-D0. */
# define GR_x_rladdr  0x1b00      /* Location to read X address bits A7-A0.
                                     Data put into D7-D0. */
# define GR_y_rhaddr  0x1bc0      /* Location to read Y address bits A9-A8. */
# define GR_y_rladdr  0x1b40      /* Location to read Y address bits A7-A0. */


# define GR_set0      0x0000      /* Address Register pair 0. */
# define GR_set1      0x0400      /* Address Register pair 1. */


# define GR_red_cmap  0x1000      /* Address to select Red Color Map */
# define GR_grn_cmap  0x1100      /* Addr for Green Color Map */
# define GR_blu_cmap  0x1200      /* Addr for Blue Color Map */

# define GR_sr_select 0x1800      /* Addr to select status register */
# define GR_cr_select 0x1900      /* Addr to select mask (color) register */
# define GR_fr_select 0x1a00      /* Addr to select function register */


/* The following are pointers to the mask(color), status, and function regs. */

# define GR_creg     (u_char *)(GR_bd_sel + GR_cr_select)
# define GR_mask     (u_char *)(GR_bd_sel + GR_cr_select)
# define GR_sreg     (u_char *)(GR_bd_sel + GR_sr_select)
# define GR_freg     (u_char *)(GR_bd_sel + GR_fr_select)


/* These assignments are for bits in the Status Register */
# define GRW0_cplane 0x00       /* Select CMap Plane number zero for R/W */
# define GRW1_cplane 0x01       /* Select CMap Plane number one for R/W */
# define GRW2_cplane 0x02       /* Select CMap Plane number two for R/W */
# define GRW3_cplane 0x03       /* Select CMap Plane number three for R/W */

# define GRV0_cplane 0x04       /* Select CMap Plane number zero for video */
# define GRV1_cplane 0x05       /* Select CMap Plane number one for video */
# define GRV2_cplane 0x06       /* Select CMap Plane number two for video */
# define GRV3_cplane 0x07       /* Select CMap Plane number three for video */
# define GR_inten     0x10      /* Enable Interrupt to start at start
                                   of next vertical retrace. Must clear bit to
                                   clear interrupts. */

# define GR_paint     0x20      /* Enable Writing five pixels in parallel */
# define GR_disp_on   0x40      /* Enable Video Display */

# define GR_vretrace 0x80       /* Unused on write. On read, true if monitor in
                                   vertical retrace. */

/* This define returns true if the board is in vertical retrace */
# define GR_retrace  (*GR_sreg & GR_vretrace)

/* The following are function register encodings */
# define GR_copy            0xCC  /* Copy data reg to Frame buffer */
# define GR_copy_invert     0x33  /* Copy inverted data reg to FB  */
# define GR_wr_creg         0xF0  /* Copy color reg to Frame buffer */
# define GR_wr_mask         0xF0  /* Copy mask to Frame buffer */
# define GRinv_wr_creg      0x0F  /* Copy inverted Creg to FB */
# define GRinv_wr_mask      0x0F  /* Copy inverted Mask to FB */
# define GR_ram_invert      0x55  /* 'Invert' color in Frame buffer */
# define GR_cr_and_dr       0xC0  /* Bitwise and of color and data regs */
# define GR_clear           0x00  /* Clear frame buffer */
# define GR_cr_xor_fb       0x5A  /* Xor frame buffer data and Creg */
```

```
#ifndef lint
static  char sccsid[] = "@(#)cgone.c 1.8 84/03/06 Copyr 1983 Sun Micro";
#endif

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

#include "cgone.h"
#include "win.h"
#if NCGONE > 0

/*
 * Sun One Color Graphics Board(s) Driver
 */

#include "../machine/pte.h"

#include "../h/param.h"
#include "../h/systm.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/proc.h"
#include "../h/buf.h"
#include "../h/conf.h"
#include "../h/file.h"
#include "../h/uio.h"
#include "../h/ioctl.h"

#include "../sun/mmu.h"
#include "../sun/fbio.h"

#include "../sundev/mbvar.h"
#include "../pixrect/pixrect.h"
#include "../pixrect/pr_util.h"
#include "../pixrect/cg1reg.h"
#include "../pixrect/cg1var.h"

#if NWIN > 0
#define CG1_OPS &cg1_ops
struct  pixrectops cg1_ops = {
        cg1_rop,
        cg1_putcolormap,
};
#else
#define CG1_OPS (struct pixrectops *)0
#endif

#define CG1SIZE (sizeof (struct cg1fb))
struct  cg1pr cgoneprdatadefault =
    { 0, 0, 255, 0, 0 };
struct  pixrect cgonepixrectdefault =
    { CG1_OPS, { CG1_WIDTH, CG1_HEIGHT }, CG1_DEPTH, /* filled in later */ 0 };


/*
```

```
 * Driver information for auto-configuration stuff.
 */
int     cgoneprobe(), cgoneintr();
struct  pixrect cgonepixrect[NCGONE];
struct  cgipr cgoneprdata[NCGONE];
struct  mb_device *cgoneinfo[NCGONE];
u_long  cgonestd[] = { 0xe8000, 0xec000, 0 };
struct  mb_driver cgonedriver = {
        cgoneprobe, 0, 0, 0, 0, cgoneintr, cgonestd, 0, CG1SIZE,
        "cgone", cgoneinfo, 0, 0, 0,
};

/*
 * Only allow opens for writing or reading and writing
 * because reading is nonsensical.
 */
cgoneopen(dev, flag)
        dev_t dev;
{
        return(fbopen(dev, flag, NCGONE, cgoneinfo));
}

/*
 * When close driver destroy pixrect.
 */
cgoneclose(dev, flag)
        dev_t dev;
{
        register int unit = minor(dev);

        if ((caddr_t)&cgoneprdata[unit] == cgonepixrect[unit].pr_data) {
                bzero((caddr_t)&cgoneprdata[unit], sizeof (struct cgipr));
                bzero((caddr_t)&cgonepixrect[unit], sizeof (struct pixrect));
        }
}

/*ARGSUSED*/
cgoneioctl(dev, cmd, data, flag)
        dev_t dev;
        caddr_t data;
{
        register int unit = minor(dev);

        switch (cmd) {

        case FBIOGTYPE: {
                register struct fbtype *fb = (struct fbtype *)data;

                fb->fb_type = FBTYPE_SUN1COLOR;
                fb->fb_height = 480;
                fb->fb_width = 640;
                fb->fb_depth = 8;
                fb->fb_cmsize = 256;
                fb->fb_size = 512*640;
                break;
                }
	case FBIOGPIXRECT: {
		register struct fbpixrect *fbpr = (struct fbpixrect *)data;
		register struct cg1fb *cg1fb =
		    (struct cg1fb *)cgoneinfo[(unit)]->md_addr;

		/*
		 * "Allocate" and initialize pixrect data with default.
		 */
		fbpr->fbpr_pixrect = &cgonepixrect[unit];
		cgonepixrect[unit] = cgonepixrectdefault;
		fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgoneprdata[unit];
		cgoneprdata[unit] = cgoneprdatadefault;
		/*
		 * Fixup pixrect data.
		 */
		cgoneprdata[unit].cgpr_va = cg1fb;
		/*
		 * Enable video
		 */
		cg1_setreg(cg1fb, CG_FUNCREG, CG_VIDEOENABLE);
		/*
		 * Clear interrupt
		 */
		cg1_intclear(cg1fb);
		break;
		}

	default:
		return (ENOTTY);
	}
	return (0);
}

/*
 * We need to handle vertical retrace interrupts here.
 * The color map(s) can only be loaded during vertical
 * retrace; we should put in ioctls for this to synchronize
 * with the interrupts.
 * FOR NOW, see comments in the code.
 */
cgoneintclear(cg1fb)
	struct  cg1fb *cg1fb;
{
	/*
	 * The Sun 1 color frame buffer doesn't indicate that an
	 * interrupt is pending on itself so we don't know if the interrupt
	 * is for our device.  So, just turn off interrupts on the cgone board.
	 * This routine can be called from any level.
	 */
	cg1_intclear(cg1fb);
	/*
	 * We return 0 so that if the interrupt is for some other device
	 * then that device will have a chance at it.
	 */
	return(0);
}
```

```
int
cgoneintr()
{
	return(fbintr(NCGONE, cgoneinfo, cgoneintclear));
}

/*ARGSUSED*/
cgonemmap(dev, off, prot)
	dev_t dev;
	off_t off;
	int prot;
{
	return(fbmmap(dev, off, prot, NCGONE, cgoneinfo, CG1SIZE));
}

#include "../sundev/cgreg.h"
	/*
	 * Note: using old cgreg.h to peek and poke for now.
	 */
/*
 * We determine that the thing we're addressing is a color
 * board by setting it up to invert the bits we write and then writing
 * and reading back DATA1, making sure to deal with FIFOs going and coming.
 */
#define DATA1 0x5C
#define DATA2 0x33
cgoneprobe(reg, unit)
	caddr_t reg;
	int	unit;
{
	register caddr_t CGXBase;
	register u_char *xaddr, *yaddr;

	CGXBase = reg;
	if (pokec((caddr_t)GR_freg, GR_copy_invert))
		return (0);
	if (pokec((caddr_t)GR_mask, 0))
		return (0);
	xaddr = (u_char *)(CGXBase + GR_x_select + GR_update + GR_set0);
	yaddr = (u_char *)(CGXBase + GR_y_select + GR_set0);
	if (pokec((caddr_t)yaddr, 0))
		return (0);
	if (pokec((caddr_t)xaddr, DATA1))
		return (0);
	peekc((caddr_t)xaddr);
	pokec((caddr_t)xaddr, DATA2);
	if (peekc((caddr_t)xaddr) == (~DATA1 & 0xFF)) {
		/*
		 * The Sun 1 color frame buffer doesn't indicate that an
		 * interrupt is pending on itself.
		 * Also, the interrupt level is user program changable.
		 * Thus, the kernel never knows what level to expect an
		 * interrupt on this device and doesn't know is an interrupt
		 * is pending.
		 * So, we add the cgoneintr routine to a list of interrupt
```

```
                 * handlers that are called if no one handles an interrupt.
                 * Add_default_intr screens out multiple calls with the same
                 * interrupt procedure.
                 */
                add_default_intr(cgoneintr);
                return (CG1SIZE);
        }
        return (0);
}

#endif
```

```
/*      @(#)skyreg.h 1.1 83/09/26 SMI   */

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Sky FFP
 */

struct  skyreg {
        u_short sky_command;
        u_short sky_status;
        union {
                short   skyu_dword[2];
                long    skyu_dlong;
        } skyu;
#define sky_data        skyu.skyu_dlong
#define sky_direg       skyu.skyu_dword[0]
        long    sky_ucode;
};

/* commands */
#define SKY_SAVE        0x1040
#define SKY_RESTOR      0x1041
#define SKY_NOP         0x1063
#define SKY_START0      0x1000
#define SKY_START1      0x1001

/* status bits */
#define SKY_IHALT       0x0000
#define SKY_INTRPT      0x0003
#define SKY_INTENB      0x0010
#define SKY_RUNENB      0x0040
#define SKY_SNGRUN      0x0060
#define SKY_RESET       0x0080
#define SKY_IODIR       0x2000
#define SKY_IDLE        0x4000
#define SKY_IORDY       0x8000
```

```
#ifndef lint
static  char sccsid[] = "@(#)sky.c 1.3 83/10/27 Copyr 1983 Sun Micro";
#endif

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 *  Sky FFP
 */
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/file.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../sun/pte.h"
#include "../sundev/mbvar.h"
#include "../sundev/skyreg.h"

/*
 * Driver information for auto-configuration stuff.
 */
int     skyprobe(), skyintr();
struct  mb_device *skyinfo[1];  /* XXX only supports 1 board */
u_long  skystd[] = { 0x2000, 0 };
struct  mb_driver skydriver = {
        skyprobe, 0, 0, 0, 0, skyintr, skystd, 0,
        sizeof (struct skyreg),
        "sky", skyinfo, 0, 0, 0
};

struct  skyreg *skyaddr;
static  int skyinit;

skyprobe(reg, unit)
        caddr_t reg;
        int unit;
{
        register struct skyreg *skybase = (struct skyreg *)reg;
        register int c;

        if ((c = peek((short *)skybase)) == -1)
                return (0);
        if (poke((short *)&skybase->sky_status, SKY_IHALT))
                return (0);
        skyaddr = (struct skyreg *)reg;
        return (sizeof (struct skyreg));
}

/*ARGSUSED*/
skyopen(dev, flag)
        dev_t dev;
        int flag;
{
	if (skyaddr == 0)
		return (ENXIO);
	if (skyinit == 2)
		u.u_skyctx.usc_used = 1;
	else if (flag & FNDELAY)
		skyinit = 1;
	else
		return (ENXIO);
	return (0);
}

/*ARGSUSED*/
skyclose(dev, flag)
	dev_t dev;
	int flag;
{

	if (skyinit == 1)
		skyinit = 2;
	u.u_skyctx.usc_used = 0;
	return (0);
}

/*ARGSUSED*/
skymmap(dev, off, prot)
	dev_t dev;
	off_t off;
	int prot;
{

	if (off)
		return (-1);
	return (getkpgmap(skyaddr) & PG_PFNUM);
}

skyintr()
{

	if (skyaddr && (skyaddr->sky_status&SKY_INTRPT)) {
		skyaddr->sky_status &= ~(SKY_INTENB|SKY_INTRPT);
		return (1);
	}
	return (0);
}

skysave()
{
	register short i;
	register struct skyreg *s = skyaddr;
	register u_short stat;

	for (i = 0; i < 100; i++) {
		stat = s->sky_status;
		if (stat & SKY_IDLE) {
			u.u_skyctx.usc_cmd = SKY_NOP;
			goto sky_save;
```

```
				}
				if (stat & SKY_IORDY)
					goto sky_ioready;
		}
		printf("sky0: hung\n");
		skyinit = 0;
		u.u_skyctx.usc_used = 0;
		return;

		/*
		 * I/O is ready, is it a read or write?
		 */
sky_ioready:
		s->sky_status = SKY_SNGRUN;	/* set single step mode */
		if (stat & SKY_IODIR)
			i = s->sky_direg;
		else
			s->sky_direg = i;

		/*
		 * Check again since data may have been a long word.
		 */
		stat = s->sky_status;
		if (stat & SKY_IORDY)
			if (stat & SKY_IODIR)
				i = s->sky_direg;
			else
				s->sky_direg = i;

		/*
		 * Read and save the command register.
		 * Decrement by 1 since command register
		 * is actually FFP program counter and we
		 * want to back it up.
		 */
		u.u_skyctx.usc_cmd = s->sky_command - 1;

		/*
		 * Reinitialize the FFP.
		 */
		s->sky_status = SKY_RESET;
		s->sky_command = SKY_START0;
		s->sky_command = SKY_START0;
		s->sky_command = SKY_START1;
		s->sky_status = SKY_RUNENB;

		/*
		 * Finally, actually do the context save function.
		 * (Unrolled loop for efficiency.)
		 */
sky_save:
		s->sky_command = SKY_NOP;	/* set FFP in a clean mode */
		s->sky_command = SKY_SAVE;
		u.u_skyctx.usc_regs[0] = s->sky_data;
		u.u_skyctx.usc_regs[1] = s->sky_data;
		u.u_skyctx.usc_regs[2] = s->sky_data;
```

```
        u.u_skyctx.usc_regs[3] = s->sky_data;
        u.u_skyctx.usc_regs[4] = s->sky_data;
        u.u_skyctx.usc_regs[5] = s->sky_data;
        u.u_skyctx.usc_regs[6] = s->sky_data;
        u.u_skyctx.usc_regs[7] = s->sky_data;
}

skyrestore()
{
        register struct skyreg *s = skyaddr;

        if (skyinit != 2) {
                u.u_skyctx.usc_used = 0;
                return;
        }
        s->sky_command = SKY_NOP;       /* set FFP in a clean mode */

        /*
         * Do the context restore function.
         */
        s->sky_command = SKY_RESTOR;
        s->sky_data = u.u_skyctx.usc_regs[0];
        s->sky_data = u.u_skyctx.usc_regs[1];
        s->sky_data = u.u_skyctx.usc_regs[2];
        s->sky_data = u.u_skyctx.usc_regs[3];
        s->sky_data = u.u_skyctx.usc_regs[4];
        s->sky_data = u.u_skyctx.usc_regs[5];
        s->sky_data = u.u_skyctx.usc_regs[6];
        s->sky_data = u.u_skyctx.usc_regs[7];
        s->sky_command = u.u_skyctx.usc_cmd;
}
```

```
/*      @(#)vpreg.h 1.3 83/08/16 SMI    */

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

/*
 * Registers for Ikon 10071-5 Multibus/Versatec interface
 * Only low byte of each word is used. (16 words total)
 * Warning - read bits are not identical to written bits.
 */
struct vpdevice {
        u_short vp_status;      /* 00: mode(w) and status(r) */
        u_short vp_cmd;         /* 02: special command bits (w) */
        u_short vp_pioout;      /* 04: PIO output data (w) */
        u_short vp_hiaddr;      /* 06: hi word of Multibus DMA address (w) */
        u_short vp_icad0;       /* 08: ad0 of 8259 interrupt controller */
        u_short vp_icad1;       /* 0A: ad1 of 8259 interrupt controller */
        /* The rest of the fields are for the 8237 DMA controller */
        u_short vp_addr;        /* 0C: DMA word address */
        u_short vp_wc;          /* 0E: DMA word count */
        u_short vp_dmacsr;      /* 10: command and status */
        u_short vp_dmareq;      /* 12: request */
        u_short vp_smb;         /* 14: single mask bit */
        u_short vp_mode;        /* 16: dma mode */
        u_short vp_clrff;       /* 18: clear first/last flip-flop */
        u_short vp_clear;       /* 1A: DMA master clear */
        u_short vp_clrmask;     /* 1C: clear mask register */
        u_short vp_allmask;     /* 1E: all mask bits */
};
/* vp_status bits (read) */
#define VP_IS8237       0x80    /* 1 if 8237 (sanity checker) */
#define VP_REDY         0x40    /* printer ready */
#define VP_DRDY         0x20    /* printer and interface ready */
#define VP_IRDY         0x10    /* interface ready */
#define VP_PRINT        0x08    /* print mode */
#define VP_NOSPP        0x04    /* not in SPP mode */
#define VP_ONLINE       0x02    /* printer online */
#define VP_NOPAPER      0x01    /* printer out of paper */
/* vp_status bits (written) */
#define VP_PLOT         0x02    /* enter plot mode */
#define VP_SPP          0x01    /* enter SPP mode */

/* vp_cmd bits */
#define VP_RESET        0x10    /* reset the plotter and interface */
#define VP_CLEAR        0x08    /* clear the plotter */
#define VP_FF           0x04    /* form feed to plotter */
#define VP_EOT          0x02    /* EOT to plotter */
#define VP_TERM         0x01    /* line terminate to plotter */

#define VP_DMAMODE      0x47    /* magic for vp_mode */

#define VP_ICPOLL       0x0C
#define VP_ICEOI        0x20
```

```
#ifndef lint
static  char sccsid[] = "@(#)vp.c 1.9 83/09/06 Copyr 1983 Sun Micro";
#endif

/*
 * Copyright (c) 1983 by Sun Microsystems, Inc.
 */

#include "vp.h"
#if NVP > 0
/*
 * Versatec matrix printer/plotter
 * dma interface driver for Ikon 10071-5 Multibus/Versatec interface
 */
#include "../h/param.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/buf.h"
#include "../h/systm.h"
#include "../h/kernel.h"
#include "../h/map.h"
#include "../h/ioctl.h"
#include "../h/vcmd.h"
#include "../h/uio.h"

#include "../sun/psl.h"
#include "../sun/mmu.h"
#include "../sundev/vpreg.h"
#include "../sundev/mbvar.h"

#define VPPRI   (PZERO-1)

struct vp_softc {
        int     sc_state;
        struct buf *sc_bp;
        int     sc_mbinfo;
} vp_softc[NVP];

#define VPSC_BUSY       0400000
/* sc_state bits - passed in VGETSTATE and VSETSTATE ioctls */
#define VPSC_MODE       0000700
#define VPSC_SPP        0000400
#define VPSC_PLOT       0000200
#define VPSC_PRINT      0000100
#define VPSC_CMNDS      0000076
#define VPSC_OPEN       0000001

#define VPUNIT(dev)     (minor(dev))

struct  buf rvpbuf[NVP];

int vpprobe(), vpintr();

u_long vpaddrs[] = { 0x400, 0x420, 0};

struct  mb_device *vpdinfo[NVP];
struct mb_driver vpdriver = {
        vpprobe, 0, 0, 0, 0, vpintr,
        vpaddrs, 0, 0,
        "vp", vpdinfo, 0, 0, 0,
};

vpprobe(reg)
        caddr_t reg;
{
        register struct vpdevice *vpaddr = (struct vpdevice *)reg;
        register int x;

        x = peek((short *)&vpaddr->vp_status);
        if (x == -1 || (x & VP_IS8237) == 0)
                return (0);
        if (poke((short *)&vpaddr->vp_cmd, VP_RESET))
                return (0);
        /* initialize 8259 so we don't get constant interrupts */
        vpaddr->vp_icad0 = 0x12;        /* ICW1, edge-trigger */
        DELAY(1);
        vpaddr->vp_icad1 = 0xFF;        /* ICW2 - don't care (non-zero) */
        DELAY(1);
        vpaddr->vp_icad1 = 0xFE;        /* IR0 - interrupt on DRDY edge */
        /* reset 8237 */
        vpaddr->vp_clear = 1;

        return (sizeof (struct vpdevice));
}

vpopen(dev)
        dev_t dev;
{
        register struct vp_softc *sc;
        register struct mb_device *md;
        register int s;
        static int vpwatch = 0;

        if (VPUNIT(dev) >= NVP ||
            ((sc = &vp_softc[minor(dev)])->sc_state&VPSC_OPEN) ||
            (md = vpdinfo[VPUNIT(dev)]) == 0 || md->md_alive == 0)
                return (ENXIO);
        if (!vpwatch) {
                vpwatch = 1;
                vptimo();
        }
        sc->sc_state = VPSC_OPEN|VPSC_PRINT | VPC_CLRCOM|VPC_RESET;
        while (sc->sc_state & VPSC_CMNDS) {
                s = splx(pritospl(md->md_intpri));
                if (vpwait(dev)) {
                        vpclose(dev);
                        return (EIO);
                }
                vpcmd(dev);
                splx(s);
        }
        return (0);
```

```
}

vpclose(dev)
        dev_t dev;
{
        register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

        sc->sc_state = 0;
}

vpstrategy(bp)
        register struct buf *bp;
{
        register struct vp_softc *sc = &vp_softc[VPUNIT(bp->b_dev)];
        register struct mb_device *md = vpdinfo[VPUNIT(bp->b_dev)];
        register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
        int s;
        int mbinfo, pa, wc;

        if (((int)bp->b_un.b_addr & 1) || bp->b_bcount < 2) {
                bp->b_flags |= B_ERROR;
                iodone(bp);
                return;
        }
        s = splx(pritospl(md->md_intpri));
        while (sc->sc_bp != NULL)               /* single thread */
                sleep((caddr_t)sc, VPPRI);
        sc->sc_bp = bp;
        vpwait(bp->b_dev);
        sc->sc_mbinfo = mbsetup(md->md_hd, bp, 0);
        vpaddr->vp_clear = 1;
        pa = MBI_ADDR(sc->sc_mbinfo);
        vpaddr->vp_hiaddr = (pa >> 16) & 0xF;
        pa = (pa >> 1) & 0x7FFF;
        wc = (bp->b_bcount >> 1) - 1;
        bp->b_resid = 0;
        vpaddr->vp_addr = pa & 0xFF;
        vpaddr->vp_addr = pa >> 8;
        vpaddr->vp_wc = wc & 0xFF;
        vpaddr->vp_wc = wc >> 8;
        vpaddr->vp_mode = VP_DMAMODE;
        vpaddr->vp_clrmask = 1;
        sc->sc_state |= VPSC_BUSY;
        splx(s);
}

/*ARGSUSED*/
vpwrite(dev, uio)
        dev_t dev;
        struct uio *uio;
{

        if (VPUNIT(dev) >= NVP)
                return (ENXIO);
        return (physio(vpstrategy, &rvpbuf[VPUNIT(dev)], dev, B_WRITE,
                    minphys, uio));
```

```
}

vpwait(dev)
	dev_t dev;
{
	register struct vpdevice *vpaddr =
	    (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
	register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];

	for (;;) {
		if ((sc->sc_state & VPSC_BUSY) == 0 &&
		    vpaddr->vp_status & VP_DRDY)
			break;
		sleep((caddr_t)sc, VPPRI);
	}
	return (0);	/* NO ERRORS YET */
}

struct pair {
	char	soft;		/* software bit */
	char	hard;		/* hardware bit */
} vpbits[] = {
	VPC_RESET,	VP_RESET,
	VPC_CLRCOM,	VP_CLEAR,
	VPC_EOTCOM,	VP_EOT,
	VPC_FFCOM,	VP_FF,
	VPC_TERMCOM,	VP_TERM,
	0,		0,
};

vpcmd(dev)
	dev_t;
{
	register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
	register struct vpdevice *vpaddr =
	    (struct vpdevice *)vpdinfo[VPUNIT(dev)]->md_addr;
	register struct pair *bit;

	for (bit = vpbits; bit->soft != 0; bit++) {
		if (sc->sc_state & bit->soft) {
			vpaddr->vp_cmd = bit->hard;
			sc->sc_state &= ~bit->soft;
			DELAY(100);	/* time for DRDY to drop */
			return;
		}
	}
}

/*ARGSUSED*/
vpioctl(dev, cmd, data, flag)
	dev_t dev;
	int cmd;
	caddr_t data;
	int flag;
{
	register int m;
	register struct mb_device *md = vpdinfo[VPUNIT(dev)];
	register struct vp_softc *sc = &vp_softc[VPUNIT(dev)];
	register struct vpdevice *vpaddr = (struct vpdevice *)md->md_addr;
	int s;

	switch (cmd) {

	case VGETSTATE:
		*(int *)data = sc->sc_state;
		break;

	case VSETSTATE:
		m = *(int *)data;
		sc->sc_state =
		    (sc->sc_state & ~VPSC_MODE) | (m&(VPSC_MODE|VPSC_CMNDS));
		break;

	default:
		return (ENOTTY);
	}
	s = splx(pritospl(md->md_intpri));
	(void) vpwait(dev);
	if (sc->sc_state&VPSC_SPP)
		vpaddr->vp_status = VP_SPP|VP_PLOT;
	else if (sc->sc_state&VPSC_PLOT)
		vpaddr->vp_status = VP_PLOT;
	else
		vpaddr->vp_status = 0;
	while (sc->sc_state & VPSC_CMNDS) {
		(void) vpwait(dev);
		vpcmd(dev);
	}
	splx(s);
	return (0);
}

/*ARGSUSED*/
vpintr()
{
	register int dev;
	register struct mb_device *md;
	register struct vpdevice *vpaddr;
	register struct vp_softc *sc;
	register int found = 0;

	for (dev = 0; dev < NVP; dev++) {
		if ((md = vpdinfo[dev]) == NULL)
			continue;
		vpaddr = (struct vpdevice *)md->md_addr;
		vpaddr->vp_icad0 = VP_ICPOLL;
		DELAY(1);
		if (vpaddr->vp_icad0 & 0x80) {
			found = 1;
			DELAY(1);
			vpaddr->vp_icad0 = VP_ICEOI;
		}
```

```
			sc = &vp_softc[dev];
			if ((sc->sc_state&VPSC_BUSY) && (vpaddr->vp_status & VP_DRDY)) {
				sc->sc_state &= ~VPSC_BUSY;
				if (sc->sc_state & VPSC_SPP) {
					sc->sc_state &= ~VPSC_SPP;
					sc->sc_state |=  VPSC_PLOT;
					vpaddr->vp_status = VP_PLOT;
				}
				iodone(sc->sc_bp);
				sc->sc_bp = NULL;
				mbrelse(md->md_hd, &sc->sc_mbinfo);
			}
			wakeup((caddr_t)sc);
		}
		return (found);
}

vptimo()
{
		int s;
		register struct mb_device *md = vpdinfo[0];

		s = splx(pritospl(md->md_intpri));
		vpintr();
		splx(s);
		timeout(vptimo, (caddr_t)0, hz);
}
#endif
```

Using ADB to Debug the UNIX Kernel

Table of Contents

Using ADB to Debug the UNIX Kernel

This document describes the use of extensions made to the UNIX† debugger *adb* for the purpose of debugging the UNIX kernel. It discusses the changes made to allow standard *adb* commands to function properly with the kernel and introduces the basics necessary for users to write *adb* command scripts which may be used to augment the standard *adb* command set. The examination techniques described here may be applied to running systems, as well as the post-mortem dumps automatically created by the *savecore*(8) program after a system crash. The reader is expected to have at least a passing familiarity with the debugger command language.

1. Introduction

Modifications have been made to the standard UNIX debugger *adb* to simplify examination of post-mortem dumps automatically generated following a system crash. These changes may also be used when examining UNIX in its normal operation. This document serves as an introduction to the use of these facilities, and should not be construed as a description of *how to debug the kernel.*

1.1. Invocation

When examining the UNIX kernel a new option, **-k**, should be used:

```
        adb -k /vmunix /dev/mem
```

This flag causes *adb* to partially simulate the Sun-2 virtual memory hardware when accessing the *core* file. In addition the internal state maintained by the debugger is initialized from data structures maintained by the UNIX kernel explicitly for debugging‡. A post-mortem dump may be examined in a similar fashion,

```
        adb -k vmunix.? vmcore.?
```

where the appropriate version of the saved operating system image and core dump are supplied in place of "?".

† UNIX is a trademark of Bell Laboratories.
‡ If the -k flag is not used when invoking *adb* the user must explicitly calculate virtual addresses. With the -k option *adb* interprets page tables to automatically perform virtual to physical address translation.

1.2. Establishing Context

During initialization *adb* attempts to establish the context of the "currently active process" by examining the value of the kernel variable *panic_regs*. This structure contains the register values at the time of the call to the *panic* routine. Once the stack pointer has been located, the command

```
	$c
```

will generate a stack trace. An alternate method may be used when a trace of a particular process is required: see section 2.3.

2. ADB Command Scripts

2.1. Extending the Formatting Facilities

Once the process context has been established, the complete *adb* command set is available for interpreting data structures. In addition, a number of *adb* scripts have been created to simplify the structured printing of commonly referenced kernel data structures. The scripts normally reside in the directory */usr/lib/adb,* and are invoked with the "**$<**" operator. A later table lists the "standard" scripts.

As an example, consider the following listing which contains a dump of a faulty process's state (our typing is shown emboldened).

```
% adb -k vmunix.3 vmcore.3
sbr 50030 slr 51e
physmem  3c0
$c
_panic[10fec](5234d) + 3c
_ialloc[16ea8](d44a2,2,dff) + c8
_maknode[1d476](dff) + 44
_copen[1c480](602,-1) +     4e
_creat() + 16
_syscall[2ea0a]() + 15e
level5() + 6c
5234d/s
_nldisp+ 175:   ialloc:     dup alloc
u$<u
_u:
_u:         pc
            4be0
_u+ 4:          d2          d3          d4          d5
            13b0        0           0           0
_u+ 14:         d6          d7
            0           2604
_u+ 1c:         a2          a3          a4          a5
            0           c7800       5a958       d7160
_u+ 2c:         a6          a7
            3e62        3e48
_u+ 34:         sr
            27000000
_u+ 38:         p0br        p0lr        p1br        p1lr
            105000      40000022    fd7f4       1ffe
_u+ 48:         szpt        sswap
            1           0
_u+ 50:         procp       ar0         comm
            d7160       3fb2        dtime^@^@^@^@^@
_u+ 158:        arg0        arg1        arg2
            1001c       -1          ffffa4
_u+ 178:        uap         qsave                   error
            2958        2eb46       1           0
```

```
_u+ 1b2:      rvl        rv2        eosys
        0          14cac      0
_u+ 1bc:      uid   gid
        49    10
_u+ 1c0:      groups
        10         -1         -1         -1
        -1         -1         -1         -1
_u+ 1e0:      ruid  rgid
        49    10
_u+ 1e4:      tsize      dsize      ssize
        7          1b         2
_u+ 344:      odsize                ossize     outime
        0          0          0
_u+ 350:      signal
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        sigmask
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0
_u+ 450:      onstack               oldmask         code
        0          80002           0
_u+ 45c:      sigstack   onsigstack
        0          0
_u+ 464:      ofile
        d66b4                 d66b4                 d66b4               0
        0          0          0          0
        0          0          0          0
        0          0          0          0
        0          0          0          0

        pofile
        0    0     0    0     0    0     0    0
        0    0     0    0     0    0     0    0
        0    0     0    0
_u+ 4c8:      cdir       rdir       ttyp       ttyd  cmask
        d44a2                 0          5c6c0      0     12

        ru & cru
```

```
_u+ 4d8:        utime                       stime
        0       0         0         35b60
_u+ 4e8:        maxrss          ixrss       idrss       isrss
        9       35        43
_u+ 4f8:        minflt          majflt          nswap
        0       5         0
_u+ 504:        inblock         oublock         msgsnd          msgrcv
        3       7         0         0
_u+ 514:        nsignals   nvcsw            nivcsw
        0       12        4
_u+ 520:        utime                       stime
        0       0         0         0
_u+ 530:        maxrss          ixrss       idrss       isrss
        0       0         0
_u+ 540:        minflt          majflt          nswap
        0       0         0
_u+ 54c:        inblock         oublock         msgsnd          msgrcv
        0       0         0         0
_u+ 55c:        nsignals   nvcsw            nivcsw
        0       0         0
0d7160$<proc
d7160:          link       rlink       addr
        590e0   0         1057f4
d716c:          upri pri   cpu  stat  time nice slp
        066     024  020   03   01    024  0
d7173:          cursig          sig
        0       0
d7178:          mask       ignore           catch
        0       0          0
d7184:          flag       uid  pgrp pid   ppid
        8001    31         2f   2f   23
d7190:          xstat      ru          poip szpt tsize
        0       0          0    1    7
d719e:          dsize      ssize       rssize      maxrss
        1b      2          5           fffff
d71ae:          swrss      swaddr           wchan           textp
        0       0          0           d8418
d71be:          p0br       xlink       ticks
        105000             0           15
d71c8:          %cpu                        ndx  idhash     pptr
        0                  6    2      d70d4
d71d4:          real itimer
        0       0          0           0
d71e4:          quota      ctx
        0       5f236
0d8418$<text
d8418:          daddr
        284     0          0           0
        0       0          0           0
        0       0          0           0
```

```
ptdaddr         size        caddr       iptr
184       7           d7160             d47e0

rssize swrss countccount      flag  slptim      poip
4      0     01    01   042  0     0
```

The cause of the crash was a "panic" (see the stack trace) due to the a duplicate inode allocation detected by the *ialloc* routine The majority of the dump was done to illustrate the use of the command scripts used to format kernel data structures. The "u" script, invoked by the command "u$<u", is a lengthy series of commands which pretty-prints the user vector. Likewise, "proc" and "text" are scripts used to format the obvious data structures. Let's quickly examine the "text" script (the script has been broken into a number of lines for convenience here; in actuality it is a single line of text).

```
./"daddr"n12Xn\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx
```

The first line produces the list of disk block addresses associated with a swapped out text segment. The "n" format forces a new-line character, with 12 hexadecimal integers printed immediately after. Likewise, the remaining two lines of the command format the remainder of the text structure. The expression "16t" causes *adb* to tab to the next column which is a multiple of 16.

The majority of the scripts provided are of this nature. When possible, the formatting scripts print a data structure with a single format to allow subsequent reuse when interrogating arrays of structures. That is, the previous script could have been written

```
./"daddr"n12Xn
+/"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn
+/"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx
```

but then reuse of the format would have invoked only the last line of the format.

2.2. Traversing Data Structures

The *adb* command language can be used to traverse complex data structures. One such data structure, a linked list, occurs quite often in the kernel. By using *adb* variables and the normal expression operators it is a simple matter to construct a script which chains down the list printing each element along the way.

For instance, the queue of processes awaiting timer events, the callout queue, is printed with the following two scripts:

callout:

```
calltodo/"time"16t"arg"16t"func"
*(.+0t12)$<callout.next
```

callout.next:

```
./D2p
*+>l
,#<l$<
<l$<callout.next
```

The first line of the script **callout** starts the traversal at the global symbol *calltodo* and prints a set of headings. It then skips the empty portion of the structure used as the head of the queue. The second line then invokes the script **callout.next** moving "." to the top of the queue ("*+" performs the indirection through the link entry of the structure at the head of the queue).

callout.next prints values for each column, then performs a conditional test on the link to the next entry. This test is performed as follows,

*+>l Place the value of the "link" in the *adb* variable "<l".

,#<l$< If the value stored in "<l" is non-zero, then the current input stream (i.e. the script **callout.next**) is terminated. Otherwise, the expression "#<l" will be zero, and the "$<" will be ignored. That is, the combination of the logical negation operator "#", *adb* variable "<l", and "$<" operator creates a statement of the form,

```
if (!link) exit;
```

The remaining line of **callout.next** simply reapplies the script on the next element in the linked list.

A sample *callout* dump is shown below.

```
% adb -k /vmunix /dev/mem
sbr 50030 slr 51e
physmem  3c0
$<callout
_calltodo:
_calltodo:  time        arg         func
d9fc4:          5           0           _roundrobin
d9f94:          1           0           _if_slowtimo
d9fd4:          1           0           _schedcpu
d9fa4:          3           0           _pffasttimo
d9fe4:          0           0           _schedpaging
d9fb4:          15          0           _pfslowtimo
d9ff4:          12          0           _arptimer
da044:          736         d7390       _realitexpire
da004:          206         d6fbc       _realitexpire
da024:          649         d741c       _realitexpire
da034:          176929      d7304       _realitexpire
```

2.3. Supplying Parameters

If one is clever, a command script may use the address and count portions of an *adb* command as parameters. An example of this is the **setproc** script used to switch to the context of a process with a known process-id;

```
0t99$<setproc
```

The body of **setproc** is

```
.>4
*nproc>l
*proc>f
$<setproc.nxt
```

while **setproc.nxt** is

```
(*(<f+0t42)&0xffff)="pid "D
,#(((*(<f+0t42)&0xffff))-<4)$<setproc.done
<l-1>l
<f+0t140>f
,#<l$<
$<setproc.nxt
```

The process-id, supplied as the parameter, is stored in the variable "<4", the number of processes is placed in "<l", and the base of the array of process structures in "<f". **setproc.nxt** then performs a linear search through the array until it matches the process-id requested, or until it runs out of process structures to check. The script **setproc.done** simply establishes the context of the process, then exits.

2.4. Standard Scripts

The following table summarizes the command scripts currently available in the directory */usr/lib/adb.*

Standard Command Scripts

Name	Use	Description
buf	***addr*$<buf**	format block I/O buffer
callout	**$<callout**	print timer queue
clist	***addr*$<clist**	format character I/O linked list
dino	***addr*$<dino**	format directory inode
dir	***addr*$<dir**	format directory entry
file	***addr*$<file**	format open file structure
filsys	***addr*$<filsys**	format in-core super block structure
findproc	***pid*$<findproc**	find process by process id
ifnet	***addr*$<ifnet**	format network interface structure
inode	***addr*$<inode**	format in-core inode structure
inpcb	***addr*$<inpcb**	format internet protocol control block
iovec	***addr*$<iovec**	format a list of *iov* structures

Standard Command Scripts

Name	Use	Description
ipreass	*addr*$<ipreass	format an ip reassembly queue
mact	*addr*$<mact	show "active" list of mbuf's
mbstat	$<mbstat	show mbuf statistics
mbuf	*addr*$<mbuf	show "next" list of mbuf's
mbufs	*addr*$<mbufs	show a number of mbuf's
mount	*addr*$<mount	format mount structure
pcb	*addr*$<pcb	format process context block
proc	*addr*$<proc	format process table entry
protosw	*addr*$<protosw	format protocol table entry
rawcb	*addr*$<rawcb	format a raw protocol control block
rtentry	*addr*$<rtentry	format a routing table entry
rusage	*addr*$<rusage	format resource usage block
setproc	*pid*$<setproc	switch process context to *pid*
socket	*addr*$<socket	format socket structure
stat	*addr*$<stat	format stat structure
tcpcb	*addr*$<tcpcb	format TCP control block
tcpip	*addr*$<tcpip	format a TCP/IP packet header
tcpreass	*addr*$<tcpreass	show a TCP reassembly queue
text	*addr*$<text	format text structure
traceall	$<traceall	show stack trace for all processes
tty	*addr*$<tty	format tty structure
u	*addr*$<u	format user vector, including pcb
uio	*addr*$<uio	format uio structure
vtimes	*addr*$<vtimes	format vtimes structure

3. Generating ADB Scripts with Adbgen

The *adbgen*(8) program allows the scripts presented earlier to be written in a way that does not depend on the structure member offsets of the items being referenced. For example, the "text" script given above depended on the fact that all the members to be printed were located contiguously in memory. Using adbgen, we could write the script as follows (again it is really on one line, but broken apart here for ease of display):

```
#include "sys/types.h"
#include "sys/text.h"

text
./"daddr"n{x_daddr,12X}n\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n\
{x_ptdaddr,X}{x_size,X}{x_caddr,X}{x_iptr,X}n\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n\
{x_rssize,x}{x_swrss,x}{x_count,b}{x_ccount,b}{x_flag,b}{x_slptime,b}{x_poip,x}{END}
```

The script starts with the names of the relevant header files, while the braces delimit structure member names and their formats. This script is then processed through *adbgen*(8) to get the *adb* script presented in the previous section. See *adbgen*(8) for a complete description of how to write *adbgen* scripts. The real value of writing scripts this way becomes apparent only with longer and more complicated scripts (for example, the "u" script). Once the scripts are written this way they can be rerun if a structure definition changes without any human effort put into offset calculations.

4. Summary

The extensions made to *adb* provide basic support for debugging the UNIX kernel by eliminating the need for a user to carry out virtual to physical address translation. A collection of scripts have been written to nicely format the major kernel data structures and aid in switching between process contexts. This has been carried out with only minimal changes to the debugger.

More work is also required on the user interface to *adb*. It appears the inscrutable *adb* command language has limited widespread use of much of the power of *adb*. One possibility is to provide a more comprehensible "adb frontend", just as *bc*(1) is used to frontend *dc*(1). Another possibility is to upgrade *dbx*(1) to understand the kernel.

The Sun UNIX File System

Table of Contents

A Fast File System for UNIX

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

1. Introduction

This paper describes the changes between the original 512 byte UNIX file system to the file system implemented with the 0.9 release of the Sun UNIX system. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

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

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput.

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

Consequently users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself*. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

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

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be

† A file system always resides on a single drive.

* The actual number may vary from system to system, but is usually in the range 5-13.

described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system.*

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

3. New file system organization

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

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

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

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because

3.1. Optimizing storage utilization

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

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

Table 1: Wasted Space as a function of Block Size

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

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

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

of the requirement that the cylinder group information must begin at a block boundary.

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

Figure 1: Example layout of blocks and fragments in a 4096/1024 file system

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

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

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The new data is written into the located piece.

3) A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine

* A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

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

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

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

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

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

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

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

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

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1) Use the available block rotationally closest to the requested block on the same cylinder.
2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.
3) If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.
4) Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

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

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to ensure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the first to get the system into a known state and the second two to ensure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Table 2: Reading Rates of the Old and New UNIX File Systems

Type of File System	Processor and Bus Measured	Speed	Read Bandwidth	% CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

Table 3: Writing rates of the old and new UNIX file systems

Type of File System	Processor and Bus Measured	Speed	Write Bandwidth	% CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is

† A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.

maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536* byte reads from contiguous tracks on the disk. The bandwidth is calculated by comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

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

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

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

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

* This number, 65536, is the maximal I/O size supported by the VAX hardware; it is a remnant of the system's PDP-11 ancestry.

the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.2. File locking

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

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

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

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* have been added similar to the scheme used by Multics [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

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

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name. To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to ensure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to ensure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

6. Software engineering

The preliminary design was done by Bill Joy in late 1980; he presented the design at The USENIX Conference held in San Francisco in January 1981. The implementation of his design was done by Kirk McKusick in the summer of 1981. Most of the new system calls were implemented by Sam Leffler. The code for enforcing quotas was implemented by Robert Elz at the University of Melbourne.

To understand how the project was done it is necessary to understand the interfaces that the UNIX system provides to the hardware mass storage systems. At the lowest level is a *raw disk*. This interface provides access to the disk as a linear array of sectors. Normally this interface is only used by programs that need to do disk to disk copies or that wish to dump file systems. However, user programs with proper access rights can also access this interface. A disk is usually formated with a file system that is interpreted by the UNIX system to provide a directory hierarchy and files. The UNIX system interprets and multiplexes requests from user programs to create, read, write, and delete files by allocating and freeing inodes and data blocks. The interpretation of the data on the disk could be done by the user programs themselves. The reason that it is done by the UNIX system is to synchronize the user requests, so that two processes do not attempt to allocate or modify the same resource simultaneously. It also allows access to be restricted at the file level rather than at the disk level and allows the common file system routines to be shared between processes.

The implementation of the new file system amounted to using a different scheme for formating and interpreting the disk. Since the synchronization and disk access routines themselves were not being changed, the changes to the file system could be developed by moving the file system interpretation routines out of the kernel and into a user program. Thus, the first step was to extract the file system code for the old file system from the UNIX kernel and change its requests to the disk driver to accesses to a raw disk. This produced a library of routines that mapped what would normally be system calls into read or write operations on the raw disk. This library was then debugged by linking it into the system utilities that copy, remove, archive, and restore files.

A new cross file system utility was written that copied files from the simulated file system to the one implemented by the kernel. This was accomplished by calling the simulation library to do a read, and then writing the resultant data by using the conventional write system call. A similar utility copied data from the kernel to the simulated file system by doing a conventional read system call and then writing the resultant data using the simulated file system library.

The second step was to rewrite the file system simulation library to interpret the new file system. By linking the new simulation library into the cross file system copying utility, it was possible to easily copy files from the old file system into the new one and from the new one to the old one. Having the file system interpretation implemented in user code had several major benefits. These included being able to use the standard system tools such as the debuggers to set breakpoints and single step through the code. When bugs were discovered, the offending problem could be fixed and tested without the need to reboot the machine. There was never a period where it was necessary to maintain two concurrent file systems in the kernel. Finally it was not necessary to dedicate a machine entirely to file system development, except for a brief period while the new file system was boot strapped.

The final step was to merge the new file system back into the UNIX kernel. This was done in less than two weeks, since the only bugs remaining were those that involved interfacing to the synchronization routines that could not be tested in the simulated system. Again the simulation system proved useful since it enabled files to be easily copied between old and new file systems regardless of which file system was running in the kernel. This greatly reduced the number of times that the system had to be rebooted.

The total design and debug time took about one man year. Most of the work was done on the file system utilities, and changing all the user programs to use the new facilities. The code changes in the kernel were minor, involving the addition of only about 800 lines of code.

Appendix A. Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when files were less stable than they should have been. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

Appendix B. References

[Accetta80] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Dion80] Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74] Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Holler73] Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78]	Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978
[Luniewski77]	Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.
[Maruyama76]	Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642
[Nevalainen77]	Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247
[Peterson83]	Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55
[Powell79]	Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42
[Porcar82]	Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.
[Ritchie74]	Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375
[Smith81a]	Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117
[Smith81b]	Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54
[Sturgis80]	Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79
[Symbolics81a]	"Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.
[Symbolics81b]	"Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.
[Thompson79]	Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979
[Thompson80]	Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???
[Trivedi80]	Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473
[White80]	White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

Sun Workstation CPU PROM Monitor

Table of Contents

The CPU PROM Monitor Commands

The central processor board (CPU) of the Sun Workstation has a set of ROM's containing a program generally known as the 'monitor'. The monitor controls the operation of the system before the UNIX kernel takes control. This document describes the PROM monitor commands. For information on the startup and boot functions of the monitor, including messages displayed, see the appendix to the *System Manager's System Installation and Maintenance Guide*: *The Sun Workstation Monitor.*

1. Command Syntax

The command format understood by the monitor is quite simple. It is:

<*verb*><*space*>*[<*argument*>]<*return*>

<*verb*> is always one alphabetic character; case does not matter.

<*space*>* means that any number of spaces is skipped here.

<*argument*>
is normally a hexadecimal number or a single letter; again, case does not matter. Square brackets '[ ]' indicate that the argument portion is optional.

<*return*> means that you should press the carriage-return key.

When typing commands, <*backspace*> and <*delete*> (also called <*rubout*>, generated by the key labelled <*backtab*> on the non-VT100 Sun keyboard) erase one character; control-U erases the entire line.

2. Syntax for Memory and Register Access

Several of the commands *open* a memory location, map register, or processor register, so that you can examine and/or modify the contents of the specified location. These commands include **a**, **d**, **e**, **l**, **m**, **o**, **p**, and **r**.

Each of these commands takes the form of a command letter, possibly followed by a hexadecimal memory address or register number, followed by a sequence of zero or more 'action specifier' arguments. The various options are illustrated below, using the **e** command as an example. You type the **boldface** parts, with a RETURN at the end of each command.

If no action specifier arguments are present, the address or register name is displayed along with its current contents. You may then type a new hexadecimal value, or simply <return> to go on the next address or register. Typing any non-hex character and RETURN will get you back to command level. For registers, 'next' means within the sequence D0-D7, A0-A6, SS, US, SF, DF, VB, SC, UC, SR, PC. For example, the following command sets consecutive locations 0x1234 and 0x1236 to the values 0x5678 and 0x0000 respectively:

```
> e1234
001234: 007F? 5678
001236: 51A4? 0
001238: C022? q
>
```

A non-hex character (such as question mark) on the command line means read-only:

```
> e1000 ?
001000: 007F
>
```

Multiple nonhex characters read multiple locations:

```
> e1000 ???
001000: 007F
001002: 0064
001004: 1234
>
```

A hex number on the command line does store-only:

```
> e1000 4567
001000 -> 4567
>
```

Multiple hex writes multiple locations:

```
> e1000 1 2 3
001000 -> 0001
001002 -> 0002
001004 -> 0003
>
```

Nonhex followed by hex reads, then stores.

```
> e1000 ? 346
001000: 007F -> 0346
>
```

Finally, reads and writes can be interspersed:

```
> e1000 ? 1 ? ? 3 4
001000: 007F -> 0001
001002: 0064
001004: 1234 -> 0003
001006 -> 0004
>
```

Spaces are optional except between two consecutive numbers. When actions are specified on the command line after the address, no further input is taken from the keyboard for that command; after executing the specified actions, a new command is prompted for. Note that these commands provide the ability to write to a location (such as an I/O register) without reading from it; and provide the ability to query a location without having to interact.

3. Command Descriptions

A [*n*][*actions*]	Open A-register *n* ($0 \leq n \leq 7$, default zero in the address space defined by the 'S' command). A7 is the System Stack Pointer; to see the User Stack Pointer, use the **r** command. For further explanation, see the section, 'Syntax for Memory and Register Access' above.
B [!][*args*]	Boot. Resets appropriate parts of the system, then bootstraps the system. This allows bootstrap loading of programs from various devices such as disk, tape, or Ethernet. Typing 'b?' lists all possible boot devices. Simply typing 'b' gives you a default boot, which is configuration dependent. For an explanation of the booting options, see the sections on 'Booting,' in the appendix to the *System Installation and Maintenance Guide* in the Sun *System Manager's Manual*. If the first character of the argument is a '!', the system reset is not done, and the bootstrapped program is not automatically executed. To execute it, use the 'C' command described below.
C [*addr*]	Continue a program. The address *addr*, if given, is the address at which execution will begin; default is the current PC. The registers will be restored to the values shown by the A, D, and R commands.
D [*n*][*actions*]	Open D-register *n* ($0 \leq n \leq 7$, default zero). For a detailed explanation, see the section, 'Syntax for Memory and Register Access' above.
E [*addr*][*actions*]	Open the word at memory address *addr* (default zero in the address space defined by the 'S' command); odd addresses are rounded down. For a detailed explanation, see the section, 'Syntax for Memory and Register Access' above.
G [*addr*][*param*]	Start the program by executing a subroutine call to the address *addr* if given, or else to the current PC. The values of the address and data registers are undefined; the status register will contain 0x2700. One parameter is passed to the subroutine on the stack; it is the address of the remainder of the command line following the last digit of *addr* (and possible blanks).
K [*number*]	If *number* is 0 (or not given), this does a 'Reset Instruction': it resets the system without affecting main memory or maps. If *number* is 1, this does a 'Medium Reset', which re-initializes most of the system without clearing memory. If *number* is 2, a hard reset is done and memory is cleared. This is equivalent to a power-on reset and causes the PROM-based diagnostics to be run, which can take ten seconds or so.
L [*addr*][*actions*]	Open the longword at memory address *addr* (default zero in the address space defined by the 'S' command); odd addresses are rounded down. For a detailed explanation, see the section, 'Syntax for Memory and Register Access' above.
M [*addr*] [*actions*]	Opens the Segment Map entry which maps virtual address *addr* (default zero) in the current context. The choice of supervisor or user context is determined by the 'S' command setting (0–3 = user; 4–7 = supervisor). See the section, 'Syntax for Memory and Register Access' above.
O [*addr*][*actions*]	Opens the byte location specified (default zero in the address space defined by the 'S' command). See the section, 'Syntax for Memory and Register Access' above. The byte versus word distinction can be a problem on the Multibus,

since some Multibus boards follow the 8086 convention for byte ordering within words, which is the reverse of the 68000 convention.

P [*addr*] [*actions*]	Opens the Page Map entry which maps virtual address *addr* (default zero) in the current context. The choice of supervisor or user context is determined by the 'S' command setting (0–3 = user; 4–7 = supervisor). With each page map entry, the relevant segment map entry is displayed in brackets. See the section, 'Syntax for Memory and Register Access' above.
R [*actions*]	Opens the miscellaneous registers (in order): SS (Supervisor Stack Pointer), US (User Stack Pointer), SF (Source Function Code), DF (Destination Function Code), VB (Vector Base), SC (System Context), UC (User Context), SR (Status Register), and PC (Program Counter). Alterations made to these registers (except SC and VC) do not take effect until the next 'C' command. For further explanation, see the section, 'Syntax for Memory and Register Access' above.
S [*number*]	Sets or queries the address space to be used by subsequent memory access commands. *number* is the function code to be used, ranging from 1 to 7. Useful values are 1 (user data), 2 (user program), 3 (memory maps), 5 (supervisor data), 6 (supervisor program). If no *number* is supplied, the current setting is printed.
U [*arg*]	The U command manipulates the on-board UARTs (serial ports) and switches the current input or output device. The argument may have the following values ('{ab}' means that either 'a' or 'b' is specified):

{ab}	Select UART a (or b) as input and output device
{ab}io	Select UART a (or b) as input and output device
{ab}i	Select UART a (or b) for input only
{ab}o	Select UART a (or b) for output only
k	Select keyboard for input
ki	Select keyboard for input
s	Select screen for output
so	Select screen for output
ks, sk	Select keyboard for input and screen for output
{ab}#	Set speed of UART a (or b) to # (such as 1200, 9600, ...)
u addr	Set virtual UART address

If no argument is specified, the U command reports the current values of the settings. If no UART is specified when changing speeds, the 'current' input device is changed.

At power-up, the following default settings are used: The default console input device is the Sun keyboard or, if the keyboard is unavailable, UART a. The default console output device is the Sun screen or, if the graphics board is unavailable, UART a. All serial ports are set to 9600 baud.