ChorusOS man pages section 7S: Devices

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3340
December 10, 1999

Adobe PostScript

Please Recycle

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME	This section gives the names of the commands or functions documented, followed by a brief description of what they do.
SYNOPSIS	This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ]	The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
. . .	Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .
|	Separator. Only one of the arguments separated by this character can be specified at time.
{ }	Braces. The options and/or arguments enclosed within braces are

interdependent, such that everything enclosed must be treated as a unit.

FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

Devices

NAME

bpf – Berkeley Packet Filter

SYNOPSIS

pseudo-device bpfilter

DESCRIPTION

The Berkeley Packet Filter provides a raw interface to data link layers in a protocol independent fashion. All packets on the network, even those destined for other hosts, are accessible through this mechanism.

The packet filter appears as a character special device such as `/dev/bpf0`, `/dev/bpf1`, and so on. After opening the device, the file descriptor must be bound to a specific network interface with the `BIOCSETIF` ioctl. A given interface can be shared by multiple listeners, and the filter underlying each descriptor will see an identical packet stream. The total number of open files is limited to the value given in the system configuration. For the binary distribution, this number is limited to `2`. For the source distribution, the maximum number of open files can be changed by modifying `NBPFILTER` in the file `iom/sys/bsd/machine/bpfilter.h`.

A separate device file is required for each minor device. If a file is in use, the open will fail and `errno` will be set to `EBUSY`.

Associated with each open instance of a bpf file is a user-settable packet filter. Whenever a packet is received by an interface, all file descriptors listening on that interface apply their filter. Each descriptor that accepts the packet receives its own copy.

Reads from these files return the next group of packets that have matched the filter. To improve performance, the buffer passed to read must be the same size as the buffers used internally by bpf. This size is returned by the `BIOCGBLEN` ioctl (see below), and can be set with `BIOCSBLEN`. Note that an individual packet larger than this size is necessarily truncated.

The packet filter will support any link level protocol that has fixed length headers. Currently, only Ethernet, SLIP, and PPP drivers have been modified to interact with bpf.

Since packet data is in network byte order, applications should use the `byteorder`(3STDC) macros to extract multi-byte values.

A packet can be sent out on the network by writing to a bpf file descriptor. The writes are unbuffered, meaning only one packet can be processed per write. Currently, only writes to Ethernet and SLIP links are supported.

IOCTLS

The `ioctl`(2POSIX) command codes below are defined in `<net/bpf.h>`. All commands require these includes:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <net/bpf.h>
```

Additionally, `BIOCGETIF` and `BIOCSETIF` require `<sys/socket.h>` and `<net/if.h>`.

In addition to `FIONREAD` and `SIOCGIFADDR`, the following commands may be applied to any open bpf file. The third argument to `ioctl`(2POSIX) should be a pointer to the type indicated.

BIOCGBLEN	(`u_int`) Returns the required buffer length for reads on bpf files.
BIOCSBLEN	(`u_int`) Sets the buffer length for reads on bpf files. The buffer must be set before the file is attached to an interface with `BIOCSETIF`. If the requested buffer size cannot be accommodated, the closest allowable size will be set and returned in the argument. A read call will result in `EIO` if it is passed a buffer that is not this size.
BIOCGDLT	(`u_int`) Returns the type of the data link layer underlying the attached interface. `EINVAL` is returned if no interface has been specified. The device types, prefixed with “DLT_”, are defined in `<net/bpf.h>`.
BIOCPROMISC	Forces the interface into promiscuous mode. All packets, not just those destined for the local host, are processed. Since more than one file can be listening on a given interface, a listener that opened its interface non-promiscuously may receive packets promiscuously. This problem can be remedied with an appropriate filter.
BIOCFLUSH	Flushes the buffer of incoming packets, and resets the statistics that are returned by `BIOCGSTATS`.
BIOCGETIF	(`struct ifreq`) Returns the name of the hardware interface that the file is listening on. The name is returned in the `if_name` field of the `ifreq` structure. All other fields are undefined.
BIOCSETIF	(`struct ifreq`) Sets the hardware interface associate with the file. This command must be performed before any packets can be read. The device is indicated by name using the `if_name` field of the `ifreq` structure. Additionally, performs the actions of `BIOCFLUSH`.

BIOCGRTIMEOUT	(`struct timeval`) Set or get the read timeout parameter. The argument specifies the length of time to wait before timing out on a read request. This parameter is initialized to zero by `open`(2POSIX), indicating no timeout.
BIOCGSTATS	(`struct bpf_stat`) Returns the following structure of packet statistics:

```
struct bpf_stat {
        u_int bs_recv;    /* number of packets received */
        u_int bs_drop;    /* number of packets dropped */
};
```

The fields are:
bs_recv

the number of packets received by the descriptor since opened or reset (including any buffered since the last read call); and
bs_drop

the number of packets which were accepted by the filter but dropped by the kernel because of buffer overflows (that is, the application's reads are not keeping up with the packet traffic).

BIOCIMMEDIATE	(`u_int`) Enable or disable "immediate mode", based on the truth value of the argument. When immediate mode is enabled, reads return immediately upon packet reception. Otherwise, a read will block until either the kernel buffer becomes full or a timeout occurs. This is useful for programs like `rarpd`, which must respond to messages in real time. The default for a new file is off.
BIOCSETF	(`struct bpf_program`) Sets the filter program used by the kernel to discard uninteresting packets. An array of instructions and its length is passed in using the following structure:

```
struct bpf_program {
        int bf_len;
        struct bpf_insn *bf_insns;
};
```

The filter program is pointed to by the `bf_insns` field while its length in units of struct bpf_insn is given by the bf_len field. Also, the actions of BIOCFLUSH are performed. See the FILTER MACHINE section for an explanation of the filter language.

BIOCVERSION

(struct bpf_version) Returns the major and minor version numbers of the filter language currently recognized by the kernel. Before installing a filter, applications must check that the current version is compatible with the running kernel. Version numbers are compatible if the major numbers match and the application minor is less than or equal to the kernel minor. The kernel version number is returned in the following structure:

```
struct bpf_version {
        u_short bv_major;
        u_short bv_minor;
};
```

The current version numbers are given by BPF_MAJOR_VERSION and BPF_MINOR_VERSION from <net/bpf.h>. An incompatible filter may result in undefined behavior (most likely, an error returned by ioctl(2POSIX) or haphazard packet matching).

BPF HEADER

The following structure is prepended to each packet returned by read(2POSIX):

```
struct bpf_hdr {
        struct timeval bh_tstamp;     /* time stamp */
        u_long bh_caplen;             /* length of captured portion */
        u_long bh_datalen;            /* original length of packet */
        u_short bh_hdrlen;            /* length of bpf header (this struct
                                              plus alignment padding */
};
```

The fields, whose values are stored in host order, are:

bh_tstamp

The time at which the packet was processed by the packet filter.

bh_caplen

The length of the captured portion of the packet. This is the minimum of the truncation amount specified by the filter and the length of the packet.

bh_datalen	The length of the packet off the wire. This value is independent of the truncation amount specified by the filter.
bh_hdrlen	The length of the bpf header, which may not be equal to `sizeof(struct bpf_hdr)`.

The `bh_hdrlen` field exists to account for padding between the header and the link level protocol. The purpose here is to guarantee proper alignment of the packet data structures, which is required on alignment sensitive architectures and improves performance on many other architectures. The packet filter insures that the `bpf_hdr` and the network layer header will be word aligned. Suitable precautions must be taken when accessing the link layer protocol fields on alignment restricted machines. (This isn't a problem on an Ethernet, since the `type` field is a `short` falling on an even offset, and the addresses are probably accessed in a bytewise fashion).

Additionally, individual packets are padded so that each starts on a word boundary. This requires that an application has some knowledge of how to get from packet to packet. The macro `BPF_WORDALIGN` is defined in `<net/bpf.h>` to facilitate this process. It rounds up its argument to the nearest word aligned value (where a word is `BPF_ALIGNMENT` bytes wide).

For example, if “p” points to the start of a packet, this expression will advance it to the next packet:

```
p = (char *)p + BPF_WORDALIGN(p->bh_hdrlen + p->bh_caplen)
```

For the alignment mechanisms to work properly, the buffer passed to `read`(2POSIX) must itself be word aligned. The `malloc`(3STDC) function will always return an aligned buffer.

FILTER MACHINE

A filter program is an array of instructions, with all branches forwardly directed, terminated by a return instruction. Each instruction performs some action on the pseudo-machine state, which consists of an accumulator, index register, scratch memory store, and implicit program counter.

The following structure defines the instruction format:

```
struct bpf_insn {
        u_short code;
        u_char  jt;
        u_char  jf;
        u_long k;
};
```

The `k` field is used in different ways by different instructions, and the `jt` and `jf` fields are used as offsets by the branch instructions. The opcodes are encoded in a semi-hierarchical fashion. There are eight classes of instructions: `BPF_LD`, `BPF_LDX`, `BPF_ST`, `BPF_STX`, `BPF_ALU`, `BPF_JMP`, `BPF_RET`, and `BPF_MISC`.

Various other mode and operator bits are `or`-ed into the class to give the actual instructions. The classes and modes are defined in `<net/bpf.h>`.

Below are the semantics for each defined bpf instruction. We use the convention that `A` is the accumulator, `X` is the index register, `P[]` packet data, and `M[]` scratch memory store. `P[i:n]` gives the data at byte offset `i` in the packet, interpreted as a word (n=4), unsigned halfword (n=2), or unsigned byte (n=1). `M[i]` gives the ith word in the scratch memory store, which is only addressed in word units. The memory store is indexed from `0` to `BPF_MEMWORDS - 1`. `k`, `jt`, and `jf` are the corresponding fields in the instruction definition. `len` refers to the length of the packet.

BPF_LD

These instructions copy a value into the accumulator. The type of the source operand is specified by an "addressing mode" and can be a constant (`BPF_IMM`), packet data at a fixed offset (`BPF_ABS`), packet data at a variable offset (`BPF_IND`), the packet length (`BPF_LEN`), or a word in the scratch memory store (`BPF_MEM`). For `BPF_IND` and `BPF_ABS`, the data size must be specified as a word (`BPF_W`), halfword (`BPF_H`), or byte (`BPF_B`). The semantics of all the recognized `BPF_LD` instructions follow.

```
BPF_LD+BPF_W+BPF_ABS  A <- P[k:4]
BPF_LD+BPF_H+BPF_ABS  A <- P[k:2]
BPF_LD+BPF_B+BPF_ABS  A <- P[k:1]
BPF_LD+BPF_W+BPF_IND  A <- P[X+k:4]
BPF_LD+BPF_H+BPF_IND  A <- P[X+k:2]
BPF_LD+BPF_B+BPF_IND  A <- P[X+k:1]
BPF_LD+BPF_W+BPF_LEN  A <- len
BPF_LD+BPF_IMM        A <- k
BPF_LD+BPF_MEM        A <- M[k]
```

BPF_LDX

These instructions load a value into the index register. Note that the addressing modes are more restrictive than those of the accumulator loads, but they include `BPF_MSH`, a hack for efficiently loading the IP header length.

```
BPF_LDX+BPF_W+BPF_IMM  X <- k
BPF_LDX+BPF_W+BPF_MEM  X <- M[k]
BPF_LDX+BPF_W+BPF_LEN  X <- len
BPF_LDX+BPF_B+BPF_MSH  X <- 4*(P[k:1]&0xf)
```

BPF_ST

This instruction stores the accumulator into the scratch memory. We do not need an addressing mode since there is only one possibility for the destination.

```
BPF_ST  M[k] <- A
```

BPF_STX

This instruction stores the index register in the scratch memory store.

```
BPF_STX  M[k] <- X
```

BPF_ALU

The alu instructions perform operations between the accumulator and index register or constant, and store the result back in the accumulator. For binary operations, a source mode is required (BPF_K or BPF_X).

```
BPF_ALU+BPF_ADD+BPF_K  A <- A + k
BPF_ALU+BPF_SUB+BPF_K  A <- A - k
BPF_ALU+BPF_MUL+BPF_K  A <- A * k
BPF_ALU+BPF_DIV+BPF_K  A <- A / k
BPF_ALU+BPF_AND+BPF_K  A <- A & k
BPF_ALU+BPF_OR+BPF_K   A <- A | k
BPF_ALU+BPF_LSH+BPF_K  A <- A << k
BPF_ALU+BPF_RSH+BPF_K  A <- A >> k
BPF_ALU+BPF_ADD+BPF_X  A <- A + X
BPF_ALU+BPF_SUB+BPF_X  A <- A - X
BPF_ALU+BPF_MUL+BPF_X  A <- A * X
BPF_ALU+BPF_DIV+BPF_X  A <- A / X
BPF_ALU+BPF_AND+BPF_X  A <- A & X
BPF_ALU+BPF_OR+BPF_X   A <- A | X
BPF_ALU+BPF_LSH+BPF_X  A <- A << X
BPF_ALU+BPF_RSH+BPF_X  A <- A >> X
BPF_ALU+BPF_NEG        A <- -A
```

BPF_JMP

The jump instructions alter flow of control. Conditional jumps compare the accumulator against a constant (BPF_K) or the index register (BPF_X). If the result is true (or non-zero), the true branch is taken, otherwise the false branch is taken. Jump offsets are encoded in 8 bits so the longest jump is 256 instructions. However, the jump always (BPF_JA) opcode uses the 32-bit k field as the offset, allowing arbitrarily distant destinations. All conditionals use unsigned comparison conventions.

```
BPF_JMP+BPF_JA          pc += k
BPF_JMP+BPF_JGT+BPF_K   pc += (A > k) ? jt : jf
BPF_JMP+BPF_JGE+BPF_K   pc += (A >= k) ? jt : jf
BPF_JMP+BPF_JEQ+BPF_K   pc += (A == k) ? jt : jf
BPF_JMP+BPF_JSET+BPF_K  pc += (A & k) ? jt : jf
BPF_JMP+BPF_JGT+BPF_X   pc += (A > X) ? jt : jf
BPF_JMP+BPF_JGE+BPF_X   pc += (A >= X) ? jt : jf
BPF_JMP+BPF_JEQ+BPF_X   pc += (A == X) ? jt : jf
BPF_JMP+BPF_JSET+BPF_X  pc += (A & X) ? jt : jf
```

BPF_RET	The return instructions terminate the filter program and specify the amount of packet to accept (i.e., they return the truncation amount). A return value of zero indicates that the packet should be ignored. The return value is either a constant (BPF_K) or the accumulator (BPF_A).

```
BPF_RET+BPF_A  accept A bytes
BPF_RET+BPF_K  accept k bytes
```

BPF_MISC	The miscellaneous category was created for anything that doesn't fit into the above classes, and for any new instructions that might need to be added. Currently, these are the register transfer instructions that copy the index register to the accumulator or vice versa.

```
BPF_MISC+BPF_TAX  X <- A
BPF_MISC+BPF_TXA  A <- X
```

The bpf interface provides the following macros to facilitate array initializers: `BPF_STMT(opcode, operand)` and `BPF_JUMP(opcode, operand, true_offset, false_offset)`.

EXAMPLES

The following filter is taken from the Reverse ARP Daemon. It accepts only Reverse ARP requests.

```
struct bpf_insn insns[] = {
         BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
         BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_REVARP, 0, 3),
         BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 20),
         BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, REVARP_REQUEST, 0, 1),
         BPF_STMT(BPF_RET+BPF_K, sizeof(struct ether_arp) +
                  sizeof(struct ether_header)),
         BPF_STMT(BPF_RET+BPF_K, 0),
 };
```

This filter accepts only IP packets between host `128.3.112.15` and `128.3.112.35`.

```
struct bpf_insn insns[] = {
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 8),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 2),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 3, 4),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 0, 3),
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 1),
        BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
        BPF_STMT(BPF_RET+BPF_K, 0),
};
```

Finally, this filter returns only TCP finger packets. We must parse the IP header to reach the TCP header. The `BPF_JSET` instruction checks that the IP fragment offset is `0` so we are sure that we have a TCP header.

```
struct bpf_insn insns[] = {
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 10),
        BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 23),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, IPPROTO_TCP, 0, 8),
        BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 20),
        BPF_JUMP(BPF_JMP+BPF_JSET+BPF_K, 0x1fff, 6, 0),
        BPF_STMT(BPF_LDX+BPF_B+BPF_MSH, 14),
        BPF_STMT(BPF_LD+BPF_H+BPF_IND, 14),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 79, 2, 0),
        BPF_STMT(BPF_LD+BPF_H+BPF_IND, 16),
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 79, 0, 1),
        BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
        BPF_STMT(BPF_RET+BPF_K, 0),
};
```

FILES

`/dev/bpf`*n*

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ioctl`(2POSIX), `byteorder`(3STDC)

McCanne, S., and Jacobson V., An efficient, extensible, and portable network monitor.

BUGS

The read buffer must be of a fixed size (returned by the `BIOCGBLEN` ioctl).

A file that does not request promiscuous mode may receive promiscuously received packets as a side effect of another file requesting this mode on the same hardware interface. This could be fixed in the kernel with additional processing overhead. However, we favor the model where all files must assume that the interface is promiscuous, and if so desired, must utilize a filter to reject foreign packets.

Data link protocols with variable length headers are not currently supported.

HISTORY

The Enet packet filter was created in 1980 by Mike Accetta and Rick Rashid at Carnegie-Mellon University. Jeffrey Mogul, at Stanford, ported the code to BSD and continued its development from 1983 on. Since then, it has evolved into the Ultrix Packet Filter at DEC, a STREAMS NIT module under SunOS 4.1, and BPF.

AUTHORS

Steven McCanne, of Lawrence Berkeley Laboratory, implemented BPF in Summer 1990. Much of the design is due to Van Jacobson.

NAME

flash – flash memory device

SYNOPSIS

`#include <sys/flashctl.h>`

FEATURES

FLASH

Note that use of this functionality requires the flash device be a supported Flite 1.2 device.

DESCRIPTION

The flash device provides both a block device interface and a character device interface.

The block device interface can be used to mount a file system stored on the flash. It must also be used to perform the `fsck_dos`(1M) utility. I/O operations using the block device interface are buffered within the `IOM` buffer cache.

The raw device interface may be used to read and write to the device as well to perform `disklabel` and `newfs_dos` operations. However, this mechanism bypasses the `IOM` buffer cache. Moreover, I/O operations on the raw device must be sector aligned (512 bytes) for both the seek pointer and the size. Otherwise an error is returned to the `read()` or `write()` system calls.

Both drivers may be used to perform the following specific `ioctl()` system calls on the flash:

FLASH_DEFRAGMENT
: is used to perform defragmentation of the flash device in order to increase the number of physical flash sectors immediately writable. The system call is as follows: `ioctl(fd,FLASH_DEFRAGMENT,(int*)&value)` where *fd* is an open file descriptor associated to the flash device and *value* is a pointer to an integer. This integer must be set prior to the call. It specifies the number of sectors which the user would like to have immediately writable. Upon completion, if no error occured, the integer pointed to by the ioctl third argument is updated with the current number of available and writeable sectors.

FLASH_SECTORDELETE
: is used to tell the flash device driver that a range of physical sectors which have been previously written, and are therefore supposed to be busy, are now available again for writing. The system call is as follows: `ioctl(fd,FLASH_SECTORETTE,&delsect)` where *fd* is an open file descriptor associated to the flash device and *delsect* is a pointer to a

structure whose type is: `flash_delsect_t`. This structure has two fields: `start` which tells the system which is the number of the first sector to delete and `count` which tells the system how many sectors should be deleted starting at sector `start`. This last `ioctl()` is used internally by the file system when a file is removed, and by the `fsck_dos`(1M) utility when it returns a sector to the free list. This `ioctl()` should be used with care.

FLASH_FORMAT	allows you to format the flash device.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Evolving

SEE ALSO

`flashdefrag`(1M), `format`(1M)

NAME

kmem, mem – Kernel space and physical memory

FEATURES

DEV_MEM

DESCRIPTION

Two devices are available on top of ChorusOS: `/dev/kmem` and `/dev/mem` . A generic driver (which works with the MEM_FLAT, MEM_PROTECTED or MEM_VIRTUAL modules) managing these devices is enabled when the `DEV_MEM` feature is set to true. They may be read, written, and sought, but not memory-mapped (the `mmap()` primitive is not supported). In addition, an `ioctl()` is available to implement two specific BSD4.4 system calls, `sysctl()` and `kvm_nlist()` .

These devices are built using the `mknod` utility as follows:

```
mknod /dev/kmem c 2 1
mknod /dev/mem c 2 0
```

`/dev/kmem` is used to access the supervisor address space, and `/dev/mem` the physical memory space. When the memory model is flat, you can access the kernel space or the physical memory using these two devices. When `VIRTUAL_ADDRESS_SPACE` is set to true, `/dev/mem` is used to access physical memory and `/dev/kmem` is used to access the supervisor space.

The `ioctl()` function on `/dev/kmem` implements the following 4.4BSD specific system calls:

```
include <sys/memio.h>
include <cx/memio.h>

int sysctl(char* mib, u_int size, void* old, size_t* oldlenp,
void* new, size_t newlen)
```

where

`mib[0] = CTL_NET`

or

`mib[0] = CTL_KERN , mib[1] = KERN_FILE`

and

`kvm_nlist(kvm_t kvmd, struct nlist* nl)`

On top of ChorusOS `sysctl()` is partially implemented. The following system I/O controls are not supported: `CTL_HW` , `CTL_VM` , `CTL_FS` , `CTL_MACDEP` , `CTL_DEBUG` and `CTL_KERN` with a mib[1] value different from KERN_FILE.

EXAMPLES

The following piece of code from 4.4BSD:

```
char  mib[6];
int   res
size_t  needed;
```

```
mib[0] = CTL_KERN;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = 0;
mib[4] = NET_RT_DUMP;
mib[5] = 0;

res = sysctl(mib, 6, NULL, &needed, NULL, 0);
```

is implemented on top of ChorusOS as follows:

```
char  mib[6];
int   res
int   kvmd;
MemIoctl ioctl_args;
size_t   needed;

mib[0] = CTL_KERN;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = 0;
mib[4] = NET_RT_DUMP;
mib[5] = 0;

ioctl_args.name = mib;
ioctl_args.namelen = 6;
ioctl_args.old = NULL;
ioctl_args.oldlenp = &needed;
ioctl_args.new = NULL;
ioctl_args.newlen = 0;
ioctl_args.retval = NULL;

kvmd = open("/dev/kmem", O_RDONLY);
/* ... */ res = ioctl(kvmd, MEMIORW, &ioctl_args);
```

The `kvm_nlist()` function used to get the values of a symbols list is translated on top of ChorusOS as follows:

```
struct nlist    nl [] = { ... };
MemSymb         ioctl_args;
int             kvmd;
kvmd = open("/dev/kmem", O_RDONLY);
/* ... */
ioctl_args.nlistp = nl;
ioctl_args.len = sizeof(nl);
res = ioctl(kvmd, MEMNLIST, &ioctl_args);
```

The `ioctl()` returns 0 when no error has occurred. It is possible that no symbol is found, in this case no value is copied out. Consequently, before calling this primitive, each symbol value must be erased as follows: `nl[i].n_value = NULL` . A non `NULL` returned value stored in `n_value` field will indicate that the symbol is found. Other `ioctl()` commands are available in the `memio.h` file for future use.

FILES

`/dev/kmem` , `/dev/mem`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`read`(2POSIX) , `write`(2POSIX) , `ioctl`(2POSIX) , `lseek`(2POSIX) , `open`(2POSIX) , `close`(2POSIX)

NAME

kmem, mem – Kernel space and physical memory

FEATURES

DEV_MEM

DESCRIPTION

Two devices are available on top of ChorusOS: `/dev/kmem` and `/dev/mem` . A generic driver (which works with the MEM_FLAT, MEM_PROTECTED or MEM_VIRTUAL modules) managing these devices is enabled when the `DEV_MEM` feature is set to true. They may be read, written, and sought, but not memory-mapped (the `mmap( )` primitive is not supported). In addition, an `ioctl( )` is available to implement two specific BSD4.4 system calls, `sysctl( )` and `kvm_nlist( )` .

These devices are built using the `mknod` utility as follows:

```
mknod /dev/kmem c 2 1
mknod /dev/mem c 2 0
```

`/dev/kmem` is used to access the supervisor address space, and `/dev/mem` the physical memory space. When the memory model is flat, you can access the kernel space or the physical memory using these two devices. When `VIRTUAL_ADDRESS_SPACE` is set to true, `/dev/mem` is used to access physical memory and `/dev/kmem` is used to access the supervisor space.

The `ioctl( )` function on `/dev/kmem` implements the following 4.4BSD specific system calls:

```
include <sys/memio.h>
include <cx/memio.h>

int sysctl(char* mib, u_int size, void* old, size_t* oldlenp,
void* new, size_t newlen)
```

where

`mib[0] = CTL_NET`

or

`mib[0] = CTL_KERN , mib[1] = KERN_FILE`

and

`kvm_nlist(kvm_t kvmd, struct nlist* nl)`

On top of ChorusOS `sysctl( )` is partially implemented. The following system I/O controls are not supported: `CTL_HW` , `CTL_VM` , `CTL_FS` , `CTL_MACDEP` , `CTL_DEBUG` and `CTL_KERN` with a mib[1] value different from KERN_FILE.

EXAMPLES

The following piece of code from 4.4BSD:

```
char  mib[6];
int   res
size_t  needed;
```

```
mib[0] = CTL_KERN;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = 0;
mib[4] = NET_RT_DUMP;
mib[5] = 0;

res = sysctl(mib, 6, NULL, &needed, NULL, 0);
```

is implemented on top of ChorusOS as follows:

```
char  mib[6];
int  res
int  kvmd;
MemIoctl ioctl_args;
size_t   needed;

mib[0] = CTL_KERN;
mib[1] = PF_ROUTE;
mib[2] = 0;
mib[3] = 0;
mib[4] = NET_RT_DUMP;
mib[5] = 0;

ioctl_args.name = mib;
ioctl_args.namelen = 6;
ioctl_args.old = NULL;
ioctl_args.oldlenp = &needed;
ioctl_args.new = NULL;
ioctl_args.newlen = 0;
ioctl_args.retval = NULL;

kvmd = open("/dev/kmem", O_RDONLY);
/* ... */ res = ioctl(kvmd, MEMIORW, &ioctl_args);
```

The `kvm_nlist()` function used to get the values of a symbols list is translated on top of ChorusOS as follows:

```
struct nlist    nl [] = { ... };
MemSymb         ioctl_args;
int             kvmd;
kvmd = open("/dev/kmem", O_RDONLY);
/* ... */
ioctl_args.nlistp = nl;
ioctl_args.len = sizeof(nl);
res = ioctl(kvmd, MEMNLIST, &ioctl_args);
```

The `ioctl()` returns 0 when no error has occurred. It is possible that no symbol is found, in this case no value is copied out. Consequently, before calling this primitive, each symbol value must be erased as follows: `nl[i].n_value = NULL` . A non `NULL` returned value stored in `n_value` field will indicate that the symbol is found. Other `ioctl()` commands are available in the `memio.h` file for future use.

FILES

`/dev/kmem` , `/dev/mem`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`read`(2POSIX) , `write`(2POSIX) , `ioctl`(2POSIX) , `lseek`(2POSIX) , `open`(2POSIX) , `close`(2POSIX)

NAME

special – special files

DESCRIPTION

This manual page describes special files related to the devices supported by ChorusOS. Special files normally reside in the `/dev` directory, which is mounted at boot time.

See `sysadm.ini`(4CC) for a list of examples of special file creation using the `mknod`(1M) command.

Generic Devices

The following special files are not specific to any hardware device and are present on all platforms.

/dev/console	Provides access to the system console regardless of its real location.
/dev/mem	Provides access to the physical memory of the system.
/dev/kmem	Provides access to the kernel memory of the system.
/dev/zero	Is a source of zeroed unnamed memory. Reads from a zero special file always return a buffer full of zeroes. The file is of infinite length. Writes to a zero special file are always successful, but the data written to it is ignored.

Peripheral Devices

By convention, special file names for storage devices follow the form `/dev/r`*suffix* for raw (character) mode and `/dev/`*suffix* for buffered (block) mode.

The *suffix* is made up of:

- A string of letters referring to the device driver name, such as `sd` for a SCSI disk, `rd` for a RAM disk, `hd` for an IDE disk or `flash` for flash,
- Followed by a digit representing the disk unit number, such as `0`, `1`, `2` and so forth,
- Terminated by a single letter referring to the partition, such as `a`, `b`, ... `h`.

Special file names for devices related to networking include:

/dev/bpf*N*	Provides access to a Berkeley Packet Filter; see `bpf`(7S).
/dev/ptyp*M*	Provides access to a master pseudo-tty device.
/dev/tty0*M*	Provides access to a tty (Teletype) character device.
/dev/ttyp*M*	Provides access to a slave pseudo-tty device.

Where *N* is a digit, such as `0`, `1`, `2` and so forth, and *M* is a digit, such as `1`, `2`, `3` and so forth.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`mknod`(1M), `sysadm.ini`(4CC)

NOTES

Unlike earlier releases that used special device driver files created on the host, this release only allows you create special files on the target.

Previous releases allowed you to create special files on the host because no `/dev` directory was available at boot time. As ChorusOS systems in this release mount a `/dev` directory at boot time, it is no longer necessary to create special files on the host.

Index