

## Section 4: Device Descriptions

This section of the manual describes the full set of Xinu device driver types available in Version 7.9 for an IBM PC fitted with a Serial Port and an ETHERNET Adapter. Each device driver implements the device independent I/O routines *open*, *close*, *read*, *write*, *seek*, *getc*, *putc*, *control*, and *init*, for one physical or pseudo-device type. While the intuitive meanings of these routines are the same across all devices, each driver defines specific, sometimes device dependent meaning to them. If a particular operation does not make sense for a given device, the driver may choose to return *SYSERR* or *OK* without taking further action.

**df(4)**

**df(4)**

## NAME

df - Xinu file system pseudo-device driver

## SYNOPSIS

**open(device, name, mode)**  
**close(device)**  
**read(device, buffer, count)**  
**write(device, buffer, count)**  
**getc(device)**  
**putc(device, char)**  
**seek(device, position)**  
**control(device, function, arg1, arg2)**  
**init(device, flag)**

## DESCRIPTION

The *df* pseudo-device driver provides access to files on a local floppy disk through the Xinu file system. The driver provides a master device (DS0) that can be opened to obtain a connection with a file, as well as driver routines for individual file pseudo-devices through which a process communicates with an open file. Files are referenced by means of device descriptors returned by OPEN(2).

The upper-half behaves as follows:

### **read(device, buffer, count)**

Reads *count* bytes of data from the file into the area starting at address *buffer*. *Read* returns a count of bytes read, which may be less than the number requested if the end of the file is reached during the operation. *Read* returns EOF if it reaches end of file without reading anything.

### **write(device, buffer, count)**

Writes *count* bytes of data from the user's *buffer* to the file starting at the current position.

### **putc(device, char)**

Writes character *char* to the file at the current position.

### **getc(device)**

Reads one character from the file and returns it to the caller. *Getc* returns EOF when it reaches the end of the file.

**df(4)**

**df(4)**

**open(DS0, filename, mode)**

Opens a Xinu file *filename* on the disk master device DS0. If successful, *open* returns the descriptor of a Xinu file pseudo-device. The string *mode* specifies the access mode for the file. Valid characters in the mode string are **r** (read access), **w** (write access), **n**(create a new file), and **o** (open an existing file). If neither **r** nor **w** is specified, both are assumed. Specifying both **n** and **o** is illegal. If neither **n** nor **o** is specified, *filename* will be created if it does not exist, otherwise the existing file *filename* will be opened.

**close(dev)**

Closes a local file pseudo-device.

**seek(dev, pos)**

Changes the current position of the Xinu file pseudo-device *dev* to *pos* (a longword). The next data transfer (e.g., read, write, getc, putc) will begin at offset *pos* (a long word).

**init(dev, flag)**

Initializes a Xinu file pseudo-device if *flag* is true.

**control(DS0, function, arg1, arg2)**

Controls the driver and provides non-transfer operations. The valid functions are:

DSKSYNC - wait for currently pending I/O requests to complete.

FLACCESS - check file *arg1* accessibility.

FLCHDSK - login a new disk in DS0

FLFORMAT - create a file system with disk ID *arg1*

FLREMOVE - delete file named by *arg1*.

FLRENAME - Rename file named *arg1* to *arg2*.

**SEE ALSO**

access(2), chdsk(2), format(2), nam(4), remove(2), rename(2)

**dos(4)**

**dos(4)**

## NAME

dos - MSDOS file system pseudo-device driver

## SYNOPSIS

```
#include <disk.h>  
#include <file.h>  
#include <mfile.h>
```

```
open(device, name, mode)  
close(device)  
read(device, buffer, count)  
write(device, buffer, count)  
getc(device)  
putc(device, char)  
seek(device, position)  
control(device, function, arg1, arg2)  
init(device, flag)
```

## DESCRIPTION

The *dos* pseudo-device driver provides access to MSDOS files on the PC. The driver provides a master device (DOS) that can be opened to obtain a connection with a file, as well as driver routines for individual file pseudo-devices through which a process communicates with an open file. Files are referenced by means of device descriptors returned by OPEN(2).

The upper-half behaves as follows:

### **read(device, buffer, count)**

Reads *count* bytes of data from the file into the area starting at address *buffer*. *Read* returns a count of bytes read, which may be less than the number requested if the end of the file is reached during the operation. *Read* returns EOF if it reaches end of file without reading anything.

### **write(device, buffer, count)**

Writes *count* bytes of data from the user's *buffer* to the file starting at the current position.

### **putc(device, char)**

Writes character *char* to the file at the current position.

### **getc(device)**

Reads one character from the file and returns it to the caller. *Getc* returns EOF when it reaches the end of the file.

**dos(4)**

**dos(4)**

**open(DOS, filename, mode)**

Opens an MSDOS file *filename* on the disk master device DOS. If successful, *open* returns the descriptor of an MSDOS file pseudo-device. The string *mode* specifies the access mode for the file. Valid characters in the mode string are **r** (read access), **w** (write access), **n** (create a new file), and **o** (open an existing file). If neither **r** nor **w** is specified, both are assumed. Specifying both **n** and **o** is illegal. If neither **n** nor **o** is specified, *filename* will be created if it does not exist, otherwise the existing file *filename* will be opened.

**close(dev)**

Closes a local file pseudo-device.

**seek(dev, pos)**

Changes the current position of the Xinu file pseudo-device *dev* to *pos* (a longword). The next data transfer (e.g., read, write, getc, putc) will begin at offset *pos*.

**init(dev, flag)**

Initializes an MSDOS file pseudo-device if *flag* is true.

**control(DOS, function, arg1, arg2)**

Controls the driver and provides non-transfer operations. The valid functions are:

FLREMOVE - delete file named by *arg1*.

FLRENAME - rename file named *arg1* to *arg2*.

FLMKDIR - create the directory *arg1*

FLMKDIRS - create the directory tree *arg1*

FLRMDIR - remove the directory *arg1*

FLRMDIRS - remove the directory tree *arg1*

**SEE ALSO**

access(2), mkdir(2), nam(4), remove(2), rename(2), rmdir(2)

**NAME**

dsk - floppy disk driver

**SYNOPSIS**

```
#include <disk.h>
```

```
read(device buffer, sector)
```

```
write(device, buffer, sector)
```

```
control(device, function, arg1, arg2)
```

```
init(device, flag)
```

**DESCRIPTION**

The *dsk* device driver provides low-level input and output for the floppy disk device. On most Xinu systems the floppy disk device is DS0. Unlike most drivers, the floppy disk driver interprets the third argument to *read* and *write* requests as a disk block number, not as a length.

The upper-half behaves as follows:

**read(device, buffer, block)**

Reads the data from the disk block number *block* into the area starting at address *buffer*.

**write(device, buffer, block)**

Writes one block of data from the user's *buffer* to the floppy disk at block number *block*.

**init(device, flag)**

Initializes the driver if *flag* is true.

**control(device, function)**

Controls the driver and provides non-transfer operations. The valid functions are:

DSKSYNC - synchronize the disk by waiting until all pending I/O completes.

**NOTES**

The *dsk* device driver has no notion of files. To perform file I/O on a Xinu disk see DF(4).

**eth(4)**

**eth(4)**

## NAME

eth - Ethernet network interface device driver

## SYNOPSIS

```
#include <ether.h>
```

```
write(device, buffer, length)
```

```
init(device, flag)
```

## DESCRIPTION

The *eth* device driver provides input and output for a 10 Mbps Ethernet local area network. The standard Xinu device name for Ethernet devices is EC0. Implemented on a 3C503 Ethernet controller, the *eth* driver handles byte input and output at the physical network level, transmitting complete packets between user processes and the device. The driver honors the following operations:

### **write(device, buffer, length)**

Writes a packet of *length* characters found in *buffer*. *Write* enforces the Ethernet minimum packet length requirement. It returns OK if the packet was accepted for transmission, SYSERR otherwise. The call to *write* will return once the packet has been placed in a port associated with the *ethout* process, but the buffer will be in use until the device finishes. The call to *write* is non-blocking, so *write* can be safely called at interrupt time. The driver calls FREEBUF(2) to dispose of the buffer once the device finishes using it.

### **init(device, flag)**

Initializes the device and driver if *flag* is true. The driver sets the device to accept only those packets addressed directly to the devices' physical address and broadcast packets (all 1's address). The state of the Ethernet Controller is saved for later restoration when *init* is called with *flag* equal to zero.

## NOTES

The *eth* driver in PCXinu 7.9 does not support a *read* function because packets arriving from the network are passed (at interrupt time) directly to the function *ni\_in*, which handles them according to type. Received ARP and RARP packets are processed immediately, while received IP packets are queued for later processing by the *ipproc* process. All other packet types are of no interest to Xinu and thus discarded.

## BUGS

The full state of the Ethernet Card is not saved prior to initialization. Restoring adapter state when Xinu exits is therefore only partially successful.

**nam(4)**

**nam(4)**

**NAME**

nam - syntactic namespace pseudo-device driver

**SYNOPSIS**

**#include <name.h>**

**open(device, name, mode)**

**init(device, flag)**

**DESCRIPTION**

The *nam* pseudo-device driver provides mapping of OPEN(2) calls to underlying devices based on name syntax. The standard Xinu name for the namespace device is NAMESPACE.

The *nam* driver provides two operations as follows:

**open(NAMESPACE, name, mode)**

Open a device given its *name* and access *mode* (see ACCESS(2) for an explanation of modes).

**init(device, flag)**

Initialize the namespace if *flag* is true by establishing a default interpretation for names.

**NOTES**

Although system calls MOUNT(2), UNMOUNT(2), NAMMAP(2), and NAMREPL(2) are intricate parts of the naming system, they are not included in the driver simply because they do not fit the read/write paradigm easily.

**SEE ALSO**

open(2), remove(2), rename(2), unmount(1)

**NAME**

pip - pipe device driver

**SYNOPSIS**

```
#include <pipe.h>
read(device buffer, length)
write(device, buffer, length)
open(device, name)
close(device)
getc(device)
putc(device, char)
init(device, flag)
```

**DESCRIPTION**

The *pipm* and *pip* device drivers, which operate as a related pair, provide a mechanism for transferring data from one process to another. Data is stored in a buffer, intermediate disk files are not used. Pipes are typically used by the shell to connect STDOUT of one process to STDIN of another process. The standard Xinu device name for the pipe master pseudo-device is PIPE, and individual pipe pseudo-devices are referenced using their device descriptor.

The drivers cooperate so that processes can open a pipe by calling OPEN(2) on the pipe master device *pipm*. If successful, the call to OPEN(2) returns the device descriptor of a *pip* pseudo-device that can be used with READ(2), WRITE(2), GETC(2) and PUTC(2) to transfer data. Finally, when finished with the pipe, a process calls CLOSE(2) on the *pip* device.

**open(device, name, mode)**

Used with the master device to open a pipe pseudo-device. *Mode* is a string that specifies whether the pipe is to be opened for reading (**ro**) or writing (**w**). A pipeline is established in two steps: process A first opens a pipe for writing, process B subsequently opens a pipe for reading. An *open* for reading will always connect to the last pipe opened for writing, or return SYSERR if no such pipe exists. Data can be written to a pipe before its other end has been opened for reading (but not vice versa). The writing process will block until another process reads from the pipe.

**read(device, buffer, length)**

Used with a *pip* pseudo-device to await the arrival of data from the pipe. *Read* returns SYSERR if *device* has not been previously opened for writing by another process.

## **pip(4)**

## **pip(4)**

### **write(device, buffer, length)**

Used with a *pip* pseudo-device to write to a pipe. The calling process will block until the data is read by another process.

### **getc(device)**

Returns a single character from the pipe *device*.

### **putc(device, char)**

Writes *char* to the pipe *device*.

### **close(device)**

The pipe *device* is only closed when *close* is called by the process which opened the pipe for reading. When called by the process which opened the pipe for writing, the pipe remains open for reading. *Close* has no effect when called by any other process.

### **init(device, flag)**

When applied to *pip* pseudo-devices with *flag* equal to true, initializes each device to mark it not in use.

## **BUGS**

The current implementation of pipes only buffers a single byte. This means that a process writing to a pipe will block for each byte transferred until another process reads that character.

**rf(4)**

**rf(4)**

**NAME** rf - remote file system pseudo-device driver (types rfm, rf)

## **SYNOPSIS**

```
#include <rfile.h>  
#include <fserver.h>  
read(device buffer, length)  
write(device, buffer, length)  
open(device, name, mode)  
close(device)  
control(device, function, arg1, arg2)  
init(device, flag)  
seek(device, offset)
```

## **DESCRIPTION**

The *rfm* and *rf* pseudo-device drivers work as a pair to provide access to remote files using a Xinu remote file server across the Internet. There is one master remote file pseudo-device (type *rfm*) for a given remote server. When users open the master remote file pseudo-device, they pass it the name of a specific file and the access mode for that file. The call to OPEN(2) returns the device descriptor of a remote file pseudo-device (type *rf*) connected to the named file.

Once opened, a user calls READ(2), WRITE(2), GETC(2), or PUTC(2), to transfer data between the user program and the remote file, or SEEK(2) to position the file pointer. The user calls CLOSE(2) to disconnect from the file. Finally, a user calls CONTROL(2) on the master pseudo-device to test file access protections, remove files, or change file names.

The standard Xinu device name for the remote file master pseudo-device is RFILESYS; individual remote file pseudo-devices are referenced by means of device descriptors. The operations on these devices are defined by:

### **open(device, name, mode)**

Opens a connection to a remote file given the file *name* and access *mode*, and returns the device descriptor used to access the file. See ACCESS(2) for valid file modes.

### **read(device, buffer, length)**

Transfers up to *length* bytes of data from a file to the user's *buffer*, and returns the number of bytes found or EOF if no more data remains in the file.

### **write(device, buffer, length)**

Writes *length* bytes of data to a file from the user's *buffer*. File length extends automatically if needed.

**rf(4)**

**rf(4)**

**close(device)**

Disconnect from a file, leaving it on secondary storage.

**control(device, function, arg1, arg2)**

Handles file manipulation other than data transfer. The possible functions are:

FLACCESS - test access (*arg1* is mode string)

FLREMOVE - remove file named by *arg1*

FLRENAME - rename file named by *arg1* to *arg2*

FLCLEAR - clear the remote file datagram port

**init(device, flag)**

When called with an *rf* device, initializes the pseudo-device data structures at system startup, if *flag* is true. Initializing the master pseudo-device has no effect.

**seek(device, offset)**

Positions the file to *offset* bytes from the beginning.

**sio(4)**

**sio(4)**

**NAME**

sio - standard input and output device

**SYNOPSIS**

**#include <sio.h>**

**open(device, name, mode)**

**close(device)**

**read(device, buffer, count)**

**write(device, buffer, count)**

**getc(device)**

**putc(device, char)**

**seek(device, position)**

**control(device, function, arg1, arg2)**

**DESCRIPTION**

The *sio* device driver implements a pseudo-device that the process can use to indirectly reference its standard input, standard output or standard error. The *sio* device performs the specified operation on the underlying device whose descriptor is `proctab[currpid].pdevs[devtab[device].dvminor]`. That is, it maps the standard I/O descriptor through the process table to find a real device descriptor to use.

The *sio* devices are called STDIN, STDOUT and STDERR.

**SEE ALSO**

`getpdev(2)`, `setpdev(2)`, `xio(4)`

sl(4)

sl(4)

## NAME

sl - serial line device driver

## SYNOPSIS

```
#include <sl.h>
```

```
read(device, buffer, count)
```

```
write(device, buffer, count)
```

```
control(device, function, arg1, arg2)
```

```
init(device, flag)
```

## DESCRIPTION

The serial line device driver is a frame-oriented device driver which is used for passing packets in standard Ethernet format over a PC COM port. After suitable Xinu configuration, the serial device (usually called SL0) can be used for SLIP access to the Internet.

### **read(SL0, buffer, count)**

Read a single packet from the serial device.

### **write(SL0, buffer, count)**

Write a single packet to the serial device.

### **control(SL0, function, arg1, arg2)**

The only *function* supported is SL\_PRNTRAW, which returns a pointer to the data field of a received packet.

### **init(SL0, flag)**

If *flag* is true, initializes the driver and sets the COM port to 2400 bps, 8 bits, no parity.

## SEE ALSO

stat(1)

**NAME**

tcp - interface to reliable 2-way Internet stream communications

**SYNOPSIS**

```
#include <network.h>
```

```
#include <tcbl.h>
```

```
#include <tcpstat.h>
```

```
read(device, buffer, length)
```

```
write(device, buffer, length)
```

```
putc(device, char)
```

```
getc(char)
```

```
open(device, name, lport)
```

```
close(device)
```

```
control(device, function, arg1, arg2)
```

**DESCRIPTION**

The *tcpl* and *tcp* devices provide an interface to the Internet TCP protocol. Programs use the master device TCP to allocate a new TCP connection endpoint or to affect global TCP configuration parameters for subsequent connections. The slave devices are particular instances of TCP servers or active TCP connections, they are referenced using their device descriptors.

The TCP devices support the following operations:

**open(TCP, name, lport)**

Allocate a new TCP connection endpoint. *Name* is a string that gives the IP address and TCP port number of the foreign endpoint in the form **i1.i2.i3.i4:t** for active opens, or the manifest constant ANYFPORT for passive opens (servers). *Lport* specifies the local port. If *lport* is ANYLPORT, Xinu will pick a port to use.

**control(device, function, arg1)**

The master device supports only the TCPC\_LISTENQ *control* function. It sets the default listen queue size for all subsequent passive opens to the integer value *arg1*. The default listen queue size is 5.

For all other TCP devices, the following control functions are defined:

TCPC_ACCEPT	For servers, accept any incoming connection attempts for a port.
TCPC_LISTENQ	Set the listen queue length to the integer value in <i>arg1</i> .
TCPC_STATUS	Return the status of a TCP connection. <i>Arg1</i> is a pointer to struct <i>tcpstat</i> which on successful return will contain the relevant statistics for the connection.

## tcp(4)

## tcp(4)

**TCPC\_SOPT/TCPC\_COPT** Set or clear TCP user options. The current options are **TCBF\_DELACK** (do delayed ACK's) and **TCBF\_BUFFER** (buffer TCP input until the full count on a read is available). *Arg1* contains the flag value(s) to set or clear.

**TCPC\_SENDURG** Send urgent data. *Arg1* is a pointer to a buffer of urgent data and *arg2* is the size (in bytes) of the data to send.

### **read(device, buffer, length)**

Read *length* characters into *buffer* from a connected TCP device. The return value is the count of characters, if non-negative. Negative values indicate various error and exceptional conditions defined as manifest constants **TCPE\_\*** in file "tcb.h".

### **write(device, buffer, length)**

Write *length* bytes from *buffer* to the connected TCP device. The return value is the number of characters written, if non-negative, and an error or exceptional condition, otherwise.

### **close(device)**

Close a TCP device. For connected devices, the call blocks until all pending data has been transmitted and acknowledged. A nonzero return value indicates that some data may have been lost.

### **SEE ALSO**

udp(4)

**NAME**

tty - console device driver

**SYNOPSIS**

```
#include <tty.h>
#include <window.h>

read(device buffer, length)
write(device, buffer, length)
open(device, name)
close(device)
getc(device)
putc(device, char)
control(device, function, arg1, arg2)
init(device, flag)
```

**DESCRIPTION**

The *tty* device driver provides input and output for a PC keyboard and screen that simulates a full-duplex ASCII terminal device. On most Xinu systems, the console device is CONSOLE.

The *tty* driver operates in one of three modes, with switching between the modes determined dynamically. In *raw* mode, it passes incoming characters to the reading process without further processing. In *cbreak* mode, the driver honors character echo, and mapping between carriage return and line feed. In *cooked* mode the driver behaves like *cbreak* mode, but also handles line editing with backspace and line kill keys. Characters are processed according to the driver mode when they arrive, and are placed in a queue from which upper-half routines extract them. Echoing, presentation of control characters, and editing are controlled by several fields in the driver control structure, and may be changed dynamically.

The upper-half routines behave as follows:

**read(device, buffer, length)**

Reads up to one line into the user's *buffer*, stopping on an END-OF-FILE or NEWLINE character, or after *length* characters have been supplied. In cooked mode, *read* blocks until a line has been typed. As a special case, if *length* is zero, the driver reads all available characters from the input buffer without waiting for the full line to be typed.

**write(device, buffer, length)**

Writes *length* characters from the user's *buffer*, mapping CARRIAGE RETURN to NEWLINE as specified by field *ocrlf* of the driver control structure. *Write* may block if the output exceeds the currently available buffer space.

**tty(4)**

**tty(4)**

**getc(device)**

Reads a single character and returns it as the function value.

**putc(device, char)**

Writes character *char*.

**open(CONSOLE, border, attr)**

Used to open a screen window pseudo-device(see WIN(4)). Returns OK if character strings *border* and *attr* are valid and the window could be opened (see WINDOW(1) and COLOR(1)). Returns SYSERR otherwise.

**close(device)**

Returns OK without taking any action.

**init(device, flag)**

Initializes the driver if *flag* is true. Note: for historical reasons, device CONSOLE is initialized to cooked mode with echo, visual control character printing, and line editing enabled; other devices may be initialized to raw mode.

**control(device, function, arg1, arg2)**

Controls the driver and provides non-transfer operations. Valid functions are:

TCNEXTC - lookahead one character without reading it

TCNEXTCI - non-blocking version of TCNEXTC

TCMODER - change the driver to raw mode

TCMODEC - change the driver to cooked mode

TCMODEK - change the driver to cbreak mode

TCMODEQ - return mode in *\*arg1*

TCMODES - switch to mode specified by *\*arg1* which must be IMCOOKED, IMCBREAK, or IMRAW

TCECHO - turn on character echo

TCNOECHO - turn off character echo

TCECHOQ - query echo mode and return in *\*arg1*

TCECHOS - turn echo on if *\*arg1* is true, otherwise turn echo off

TCICHARS - return a count of characters in the input buffer

TCFLUSH - flush the input buffer

TCINT - make calling process receive interrupt messages

TCINTQ - query pid of process receiving interrupt messages

TCINTS - set pid of process to receive interrupt message to *\*arg1*

TCNOINT - turn off interrupt message processing

TCCLEAR - clear screen

TCCURPOS - position cursor (*arg1* contains position string)

TCATTR - change screen attributes (*arg1* contains attribute string)

**SEE ALSO**

win(4)

**udp(4)**

**udp(4)**

## NAME

udp - UDP-level Internet interface pseudo-device driver (types dgm, dg).

## SYNOPSIS

```
#include <network.h>  
read(device buffer, length)  
write(device, buffer, length)  
open(device, name)  
close(device)  
control(device, function, arg1)  
init(device, flag)
```

## DESCRIPTION

The *dgm* and *dg* device drivers, which operate as a related pair, provide a network interface at the IP datagram level. They accept datagrams from user processes and send them out on the Internet, or receive datagrams from the Internet and deliver them to user processes. The standard Xinu device name for the datagram master pseudo-device is UDP, and individual connection pseudo-devices are referenced by using their device descriptors.

The drivers cooperate so that users can initiate a connection by calling OPEN(2) on the datagram master device. If successful, the call to OPEN(2) returns the device descriptor of a *dg* pseudo-device that can be used with READ(2) or WRITE(2) to transfer data. Finally, when finished with the connection, the user process calls CLOSE(2) on the connection device descriptor.

The *dgm* driver consists of routines that implement OPEN(2) and CONTROL(2), while the *dg* driver consists of routines for READ(2), WRITE(2), CLOSE(2), and CONTROL(2). Primitives READ(2) and WRITE(2) operate in one of two basic modes. Either they transfer data in Xinugram format complete with an address header, or they transfer just the data portion of the datagram.

### **open(UDP, name, mode)**

Used with the master device to open a datagram pseudo-device. *Name* is a string that gives an IP address and UDP port number in the form **i1.i2.i3.i4:u**.

## udp(4)

## udp(4)

### control(device, function, arg1)

No control operations are supported for the master device.

Used with a pseudo-device descriptor to set the transfer mode. The valid operations include DGCLEAR, which clears any UDP datagrams that happen to be in the receive queue, and DGSETMODE, which sets the pseudo-device mode. The mode argument *arg1* is composed of a word in which the first two bits control the transfer mode and the sixth bit controls timeout. The symbolic constants for these bits are:

DG\_NMODE (001) - Normal mode

DG\_DMODE (002) - Data-only mode

DG\_TMODE (004) - Timeout all reads

DG\_CMODE (010) - Generate UDP checksums (default on)

### read(device buffer, length)

Used with a *dg* pseudo-device to await the arrival of a UDP datagram and transfer it to the user in the form of a Xinugram.

### write(device, buffer, length)

Used with a *dg* pseudo-device to transfer a Xinugram into a UDP datagram and send it on the Internet.

### close(device)

Closes a *dg* pseudo-device.

### init(device, flag)

When applied to *dg* pseudo-devices with *flag* equal to true, initializes each device to mark it not in use.

### SEE ALSO

tcp(4)

### BUGS

The DG\_TMODE timeout value cannot be changed dynamically.

**NAME**

win - window driver

**SYNOPSIS**

```
#include <tty.h>
```

```
#include <window.h>
```

```
open(device, border, attrib)
```

```
close(device)
```

```
read(device, buffer, length)
```

```
write(device, buffer, length)
```

```
getc(device)
```

```
putc(device, char)
```

```
control(device, function, arg1, arg2)
```

```
init(device, flag)
```

**DESCRIPTION**

The *win* device driver provides input and output for windows that simulate a full-duplex ASCII terminal device. Windows are created by calling *open* on the *tty* device CONSOLE.

The *win* driver operates in one of three modes, with switching between the modes determined dynamically. In *raw mode*, it passes incoming characters to the reading process without further processing. In *cbreak mode*, character echo, and mapping between carriage return and line feed is provided. In *cooked mode* the driver behaves like *cbreak mode*, but also handles line editing with backspace and line kill keys. Characters are processed according to the driver mode when they arrive, and are placed in a queue from which upper-half routines extract them. Echoing, presentation of control characters, and editing are controlled by several fields in the driver control structure, and may be changed dynamically.

The upper-half routines behave as follows:

**read(device, buffer, length)**

Reads up to one line into the user's *buffer*, stopping on an END-OF-FILE or NEWLINE character, or after *length* characters have been supplied. In cooked mode, *read* blocks until a line has been typed. As a special case, if *length* is zero, the driver reads all available characters from the input buffer without waiting for the full line to be typed.

**write(device, buffer, length)**

Writes *length* characters from the user's *buffer*, mapping CARRIAGE RETURN to NEWLINE as specified by field *ocrlf* of the driver control structure. Write may block if the output exceeds the currently available buffer space.

**win(4)**

**win(4)**

**getc(device)**

Reads a single character and returns it as the function value.

**putc(device, char)**

Writes character *char*.

**open(CONSOLE, border, attr)**

*Open* allocates a new window specified by *border* and *attr* (see WINDOW(1)) and returns its device descriptor.

**close(device)**

Closes the window and deallocates associated data structures.

**init(device, flag)**

Initializes the window driver if *flag* is true.

**control(device, function, arg1)**

Controls the driver and provides non-transfer operations. The valid functions are:

TCNEXTC - lookahead one character without reading it

TCNEXTCI - non-blocking version of TCNEXTC

TCMODER - change the driver to raw mode

TCMODEC - change the driver to cooked mode

TCMODEK - change the driver to cbreak mode

TCMODEQ - return mode in *\*arg1*

TCMODES - switch to mode specified by *\*arg1* which must be IMCOOKED, IMCBREAK, or IMRAW

TCECHO - turn on character echo

TCNOECHO - turn off character echo

TCECHOQ - query echo mode and return in *\*arg1*

TCECHOS - turn echo on if *\*arg1* is true, otherwise turn echo off

TCICHARS - return a count of characters in the input buffer

TCFLUSH - flush the input buffer

TCINT - make calling process receive interrupt messages

TCINTQ - query pid of process receiving interrupt messages

TCINTS - set pid of process to receive interrupt message to *\*arg1*

TCNOINT - turn off interrupt message processing

TCCLEAR - clear screen

TCCURPOS - position cursor (*arg1* contains position string)

TCATTR - change screen attributes (*arg1* contains attribute string)

**SEE ALSO**

tty(4), window(1)

**xio(4)**

**xio(4)**

**NAME**

xio - Xinu I/O device driver

**SYNOPSIS**

**#include <sio.h>**

**open(device, name, mode)**

**DESCRIPTION**

The *xio* pseudo-device driver provides a mechanism for opening devices and recording their device descriptors in the process table. Recorded devices are closed automatically when the process is terminated. A call to *open* on device *xio* will open a device *name* in mode *mode* on the NAMESPACE device. The device descriptor of the device actually opened is returned after being recorded in the process table.

**SEE ALSO**

getpdev(2), setpdev(2), sio(4)

This page is empty.