Appendix 2

PC-Xinu Programmer's Manual

Version 6pc

(XINU IS NOT UNIX)

The PC-Xinu Programmer's Manual contains a description of the PC-Xinu software. It has been divided into three sections, following the style of the *UNIX Programmer's Manual*. This introduction explains how to compile, link, and execute a program that uses PC-Xinu. It also contains a set of informal implementation notes that characterize PC-Xinu.

The body of the manual gives a terse description of PC-Xinu procedures and the details of their arguments – it is intended as a quick reference for programmers, not as a way to learn PC-Xinu. Section 1 describes the *config* program that runs on the host operating system. Section 2 describes PC-Xinu system procedures that programs call to invoke operating system services. (The system calls supported by PC-Xinu include all system calls available for the original LSI-11 version plus a few additional PC-specific services). Section 3 describes additional procedures available from the PC-Xinu library. From the programmer's point of view, there is little distinction between library routines and system calls; the distinction is preserved here to help beginners understand the subtlities.

As in the UNIX manual, each page describes one command, system call, or library routine; section numbers appear in the page footer as ''(digit)'' following the name of the program. Within a section all pages are arranged in alphabetical order. References have the same form as headers (e.g., ''getc(2)'' refers to the page for ''getc'' in section 2). Related commands are sometimes mentioned on one page (which may make it difficult for beginners to find them).

A Short Introduction To PC-Xinu and the Host Development Software

How to Use PC-Xinu

Overview. PC-Xinu was developed using MS-DOS on a PC, which is the same machine on which PC-Xinu runs. To run PC-Xinu, you create, compile and link a source file under MS-DOS, and run the resulting executable file.

Host Development environment. To run the software you will need an IBM PC/XT/AT or compatible running MS-DOS version 2.1 or higher (version 3.0 or higher recommended). Also recommended is a hard disk system with at least 5 megabyes storage capacity. The MS-DOS host development system contains a C compiler, assembler, linker, and 'make' facility. The Microsoft versions of these are called *MSC*, *masm*, *LINK*, and *make*, respectively. The Turbo C compiler *tcc* may be used in place of the Microsoft C compiler. (These names may be typed in uppercase as well.) The versions used for development of PC-Xinu are given in the following table:

Program	Version
MSC	4.00
MASM	4.00
MAKE	4.02
LINK	3.06
TCC	1.00

Compiling programs. The command *msc* invokes the Microsoft C compiler, and translates a C program with extension *.c* into an object module with extension *.obj*. The Turbo C compiler command is *tcc*. *Link* is used to combine this object module and the PC-Xinu library *xinu.lib* into an executable program. There must be one external routine named *xmain*, which becomes the user process executed after PC-Xinu initialization.

Running and terminating PC-Xinu. When you execute your program, the system prints initialization information, after which you are asked to press any key to begin execution of the program you created. Even if your program terminates, PC-Xinu will still be active, since there are a number of server processes which under ordinary circumstances do not get killed. To return to the host operating system, key *Ctrl-Break*. In some cases your program may crash, and you may find it necessary to reboot the entire system. Debugging a crashed system is not easy, since the normal MS-DOS debugging tools are not effective when used with interrupt-driven software.

An Example

Create a C program. For example, here is a C program *example.c* that prints the string ‘‘Hello world.’’ on the console terminal and exits. (*Printf* is a system (library) procedure that prints formatted strings on the console; other system commands are described in sections 2 and 3 of this manual):

```
/* example.c example C program (stored in file example.c) */

#include <conf.h>
#include <kernel.h>

xmain()
{
    printf("Hello world.\n");
}
```

Compile, link and run. Compile the program, link it with the PC-Xinu library, and run it with the following commands:

```
msc example /FPa /I . ;
link example,,nul,..\lib\xinu;
example
```

If you are using the Turbo C compiler, use the following commands (the link command has been shown on two lines to fit it into the text; enter only one line when typing it to the computer):

```
tcc -c -DTURBOC example
link \turboc\lib\c0s+example,example,nul,
     ..\lib\xinu+\turboc\lib\cs+\turboc\lib\maths
example
```

The compiler will compile the C program *example.c*, producing an object module *example.obj*. When using the Microsoft C compiler, the switch */FPa* directs the compiler to use the software arithmetic package, and the switch */I* followed by a dot informs the compiler to search for #include files in the current directory. The switch -DTURBOC used with Turbo C directs the compiler to define the symbol *TURBOC*. We define the *TURBOC* symbol to make it possible to select one version of code for either the Turbo C or Microsoft C compilers.

The linker will combine the *example.obj* file with library routines in *xinu.lib* as well as some standard Microsoft C or Turbo C libraries to produce an executable file *example.exe*. The Microsoft linker looks at the MS-DOS *LIB* environment variable to find the location of the default Microsoft C library used to create the executable file. In

the Microsoft C development of PC-Xinu, we set the *LIB* environment variable as follows:

```
set LIB=c:\msc\lib
```

The Turbo C compiler uses the *turboc.cfg* file to identify default directories.

Invoking *example* runs the program *example.exe*. When your program begins execution you will see a few PC-Xinu system startup messages followed by the program output. When you key *Ctrl-Break* to terminate your program, you will see a system termination message. The output is:

```
Initializing . . .
Disk read error 80H reading drive 0

PC-Xinu Version 6pc (1-Dec-87)
64484 real mem
16360 base addr
48124 avail mem

Hit any key to continue . . .

Hello world.
^@

-- system halt --

PC-Xinu terminated with 3 processes active
Returning . . .
```

The ''Disk read error'' message comes from having no disk in drive 0 (drive A); if you have a disk in drive 0, you will not get this message. After responding to the request to ''Hit any key'', the screen will clear and the program will begin execution. In this case, the ''Hello world.'' message will be printed and the system will appear to hang. The *Ctrl-Break* key is used to return control to MS-DOS and echoes as ''^@''; the messages beginning with ''-- system halt --'' will be printed before returning to MS-DOS.

PC-Xinu Directory Organization

The PC-Xinu software distribution disks available from the publisher contain all the software from this book. Source files are included. The disks come with instructions needed to install and use the software.

Directories on the distribution disks are described below. Usually, the PC-Xinu software is rooted in the directory *\pcxinu* , although this is not necessary.

```
Disk 1:
  src           All system source files

Disk 2:
  config        PC-Xinu configuration and sources

  examples      Example programs from Chapter 1

  util          Undocumented disk file utilities

Disk 3:
  test          Sample test files (sources and executables)

  lib           The PC-Xinu library

Disk 4:
  PC-Xinu formatted disk (not readable by MS-DOS)
```

Xinu Design Notes

(Updated 1/82, 3/82, 11/82, 3/83, 7/87)

These are the notes kept during implementation; they are not designed as an accurate introduction to PC-Xinu. In particular, deferred operations (e.g., deferred process termination) have disappeared from the book version along with the *pmark* process table entry even though they remain in version 5.

Some quick ideas:

- There are multiple processes.
- A process is known by its process id.
- The process id is an index into the process table.
- There are counting semaphores.
- A semaphore is known by its index in the semaphore table.
- There is a line time clock.
- The system schedules processes based on priority.
- The system supports I/O.
- For tty I/O, characters are queued on input and output. The normal mode includes echoing, erasing backspace, etc.
- There is support for non-overlapping screen windows which behave just like tty devices.
- There is a file system that supports concurrent growth of files without preallocation; it has only a single-level directory structure.
- MS-DOS files can be accessed directly through PC-Xinu I/O calls.
- There is a one-word message passing mechanism (a ''word'' is large enough to hold a pointer).
- There is support for self-initializing routines (memory marking) that makes it unnecessary for the kernel to call initialization routines explicitly.
- Processes can create processes, kill processes, suspend processes, restart processes, and change the priority of processes.
- There is no real memory management, but there are primitives for acquiring and returning memory from the global pool, and a way to suballocate smaller pools of fixed-size buffers.
- There is a configuration program, *config*, to generate a PC-Xinu system according to specifications given in the file *pcxconf*.

Discussion of implementation:

0. Files. The system sources are organized as a set of procedures. For the most part, there is a file for each system call (e.g., *chprio.c* for the system call *chprio*). In addition to the system call procedures, a file may contain utility functions needed by

that system call. Files which do not correspond to system calls, utility procedures or device driver routines are listed below (in order of their appearance in the book) along with a brief description of each:

kbdio.h	BIOS constants and function declarations for keyboard routines.
kbdio.asm	Access to keyboard interrupt functions.
vidio.h	BIOS constants and function declarations for video routines.
vidio.asm	Access to video interrupt functions.
dskio.h	BIOS constants and function declarations for floppy disk routines.
dskio.asm	Access to disk interrupt functions.
dio.c	Sector-oriented disk read/write routines.
q.h	Queue data structure declaration (see below); defined macros for queue predicates.
queue.c	Queue manipulation routines.
insert.c	Insert a process into a queue in key order.
getitem.c	Remove the first or last item from a list.
newqueue.c	Initialize a new list in the queue structure.
proc.h	Process table entry structure declaration; state constants.
resched.c	Almost the inner-most routine (rescheduler). It selects the next process to run from the ready queue and fixes up the state. Calls *ctxsw* to switch contexts.
cxtsw.asm	The routine that actually changes the executing process into another one. A very small piece of assembler code with only one trick: when a process is saved, the execution address at which it restarts is actually the instruction following the call to ctxsw.
ready.c	Make a process eligible for CPU service.
eidi.asm	Procedures to enable/disable CPU interrupts.
xdone.c	Terminate PC-Xinu and return to MS-DOS.
kernel.h	General symbolic constants; miscellaneous defined macros.
userret.c	The routine to which user processes return (i.e. exit) if they ever do. Care should be taken so that *userret* never exits; it must kill the process that runs it because there is no legal return frame on the stack.
sem.h	Semaphore table entry structure declaration; semaphore constants.
mem.h	Memory management free list format declarations.
io.h	General user-level I/O definitions.
intmap.asm	Interrupt dispatch table and routine.
xeidi.asm	Routines to enable/disable PC-Xinu interrupt handling.
bios.h	Low-level BIOS definitions.
insertd.c	Procedure to insert process in sleep queue (delta list).
sleep.h	Definitions for real-time delay software.
clkint.c	Clock interrupt service routine.
wakeup.c	Procedure to remove awakened processes from the sleep queue.
ssclock.c	Procedures to stop/start defer mode for the real-time clock.
clkinit.c	Routine to initialize the real-time clock.

conf.h	Generated constants including I/O and size constants; do not edit directly.
ioerr.c	Generic I/O procedure which always returns *SYSERR*.
ionull.c	Generic I/O procedure which always returns *OK*.
conf.c	Generated file of initialized variables; do not edit directly.
map.c	Procedures to initialize and restore *intmap* entries.
tty.h	*Tty* line discipline control block and buffers, excluding sizes.
writcopy.c	Memory-to-memory copy for transfer from a user buffer to the tty output buffer.
readcopy.c	Memory-to-memory copy for transfer from the tty input buffer to a user buffer.
initiali.c	All external (global) variables, the null process (process 0, see below), and the high-level system initialization routine (e.g., to craft the process table entry for process 0).
stack.asm	Routines to manage stack limit checking.
window.h	Window control block and buffers, similar to *tty.h*.
lwbord.c	Parse the window border string and return window coordinates.
lwattr.c	Parse the window attribute string and return the modified attributes.
pcscreen.c	Routines to access BIOS screen handling functions.
wputcsr.c	Position the cursor relative to window coordinates.
mark.h	Memory marking table declarations.
bufpool.h	Buffer pool constants and format.
ports.h	Definitions for communication ports (queued interprocess rendezvous points).
ptclear.c	Utility routine to clear a message port.
disk.h	Disk driver control block.
iblock.h	Layout of index block (i-block).
dir.h	Layout of disk directory block.
file.h	Definitions of variables and constants used by the local file system routines.
doscall.c	Routines to access low-level MS-DOS file functions.
mffile.h	Definitions for MS-DOS file devices.
doprnt.c	Output formatter for *printf, kprintf* etc.
butler.h	Definitions for the **BUTLER** process.
butler.c	Code for the **BUTLER** process.
cbrkint.c	*Ctrl-Break* interrupt handler.
psnap.c	Process snapshot routine.
tsnap.c	*Tty* queue snapshot routine.
dsnap.c	Disk queue snapshot routine.
pcxconf	The file of device and constant information as edited by the user to describe the hardware; the *config* program takes this file and produces *conf.c* and *conf.h* files.

1. Process states. Each process has a state given by the *pstate* field in the process table entry. The state values are given by symbolic constants PRxxxx. PRFREE means that the process entry is unused. PRREADY means that the process is linked into the ready list and is eligible for the CPU. PRWAIT means that the process is waiting on a semaphore (given by *psem*). PRSUSP means that the process is in hibernation; it is not on any list. PRSLEEP means that the process is in the queue of sleeping processes waiting to be awakened. PRCURR means that the process is (the only one) currently running. The currently running process is NOT on the ready list.
2. Semaphores. Semaphores reside in the array *semaph*. Each entry in the array corresponds to a semaphore and has a count (*scount*), and state (*sstate*). The state is SFREE if the semaphore slot is unassigned, SUSED if the semaphore is in use. If the count is -*p* then the *sqhead* and *sqtail* fields point to a FIFO queue of *p* processes waiting for the semaphore. If the count is nonnegative *p*, then no processes are waiting. More about the head and tail pointers below.
3. Suspension. Suspended processes are forbidden from using the CPU. They may remain on semaphore/sleep queues until they are to be moved to the ready queue. Suspending a process that is already on the ready queue will remove it. Suspending the current processes forces it to give up the CPU. A call to *ready*(*p*), where *p* has been marked suspended, will place it on the ready queue.
4. Sleeping. When a process calls *sleep*(*n*) it will be delayed *n* seconds. This is achieved by placing the process on a queue of jobs ordered by wakeup time and relinquishing the CPU. With each clock tick, the real-time clock will interrupt the CPU and cause a clock interrupt routine to be called. Ticks occur approximately 18.2 times per second. The interrupt handler moves processes back to the ready queue when their wakeup time has been reached. Notice that a process may put itself, but no one else, to sleep.
5. Queues and ordered lists. There is one data structure for all heads, tails, and elements of queues or lists of processes: *q*[]. The first NPROC entries in *q* (0 to NPROC-1) correspond to the NPROC processes. If one wants to link process *i* onto a queue or list, then one uses *q*[*i*].*qnext* and *q*[*i*].*qprev* as the forward and backward pointers.

   The remaining entries in *q* are used for the heads and tails of lists. The integer *nextqueue* always points to the next available entry in *q* to assign. When initialize builds the heads and tails of various lists, it assigns entries in *q* sequentially. Thus, the *sqhead* and *sqtail* fields of a semaphore are really the indices of the head and tail of the list in *q*. The advantage of keeping all heads and tails in the same data structure is that enqueuing, dequeueing, testing for empty/nonempty, and removing from the middle (eg., when a process is killed) are all handled by a small set of simple procedures (files *queue.c* and *q.h*). An empty queue has the head and tail pointing to each other. Since all real items have index less than NPROC, testing whether a list is empty becomes trivial. In addition to FIFO queues, *q* also contains

ordered lists based on an integer kept in the *qkey* field. For example, processes are inserted in the ready list (head at position *q*[*rdylist*]) based on their priority. They are inserted in the sleep list based on wakeup time. Ordered lists are always in ascending order with the inserted item stuck in *before* those with an equal key. Thus, processes are removed from the ready list from the tail to get the highest priority process. Also, processes of equal priority are scheduled round robin. Since the sleep queues are serviced from the smallest to largest keys, items are removed from the head of the queue (equal keys do not matter for sleeps).

6. Process 0. Process 0 is a null process that is always available to run or is running. Care must be taken so that process 0 never executes code that could cause its suspension (e.g. it cannot wait for a semaphore). Since Process 0 may be running during interrupts, this means that interrupt code may never wait for a semaphore. Process 0 initializes the system, creates the first user process, starts it executing the main program, and goes into an infinite loop waiting until an interrupt. Because its priority is lower than that of any other process, the null process loop executes only when no other process is ready.

Section 1: Development Commands

This section of the manual describes the PC-Xinu development commands that run under the host operating system and that are used to create a PC-Xinu executable program. The only command described here is the PC-Xinu configuration generator *config*. Consult vendor documentation for information about the C compiler, assembler, library, and linker utilities.

NAME

config – PC-Xinu configuration generator

SYNOPSIS

config [-f ifile] [-c cfile] [-h hfile]

DESCRIPTION

Config generates the device configuration files *conf.c* and *conf.h* from the configuration description file *pcxconf.*

If the *-f ifile* command-line parameter is present, the input file is taken from the file *ifile* instead of the default *pcxconf.* If the *-c cfile* parameter is present, *cfile* will be used in place of *conf.c* for output. If the *-h hfile* parameter is present, *hfile* will be used in place of *conf.h* for output.

The configuration description file-- which is ordinarily *pcxconf* --is divided into three sections, separated from each other with a '%' character, following the format below:

```
device class and type declarations
%
device definitions
%
configuration constants
```

White space and standard C comments of the form /*...*/ are ignored in the first two sections.

A device class and type declaration has the form

```
class: [ on typ ] [ -op function ]* [ addr= value ]*
```

where items in brackets are optional, and an asterisk '*' means that zero or more fields may be present. The *class* field is a device class identifier (usually lowercase), and the *typ* field is a device type identifier within that device class (usually uppercase). Identifiers must conform to C identifier conventions. A device class declaration may have more than one device type; in this case, the device class identifier appears only once, and each type declaration must begin with an 'on typ' field. If there is only one type corresponding to a device class, the 'on typ' field may be omitted.

An *-op* field in a device class declaration may be one of the following:

```
-i -o -c -r -w -s -g -p -n -iint -oint
```

The *function* identifiers following these operations are device-specific functions

corresponding to the abstract operations *dvinit*, *dvopen*, *dvclose*, *dvread*, *dvwrite*, *dvseek*, *dvgetc*, *dvputc*, *dvcntl*, and the interrupt service routines *dviint*, and *dvoint*, respectively, as defined in the device switch table *devtab*.

An *addr=* field in a device class declaration may be one of the following address designators:

name= port= ivec= ovec=

The *value* fields following these address designators are device-specific addresses corresponding to the fields *dvname*, *dvport*, *dvivec* and *dvovec*, respectively, in *devtab*. The *value* argument may be a constant identifier, an integer (in decimal, octal or hexadecimal C format) or a string enclosed in double quotes which evaluates to a constant, except that the value corresponding to *name=* must be either an identifier or a quoted string of length less than 10.

The values given in this declaration section will be used as defaults when defining actual devices.

If a operation *-op* is missing in a device class declaration, it defaults to *ioerr*. Similarly, if an address designator *addr=* is missing, it defaults to zero (except that if a name designator *name=* is missing, it defaults to the empty string).

The *device definition* section is used to create device entries in the device switch table *devtab* in the file *conf.c*. Each device definition is given a device number which are assigned consecutively starting with zero. In addition, the number of devices corresponding to each device class is counted and is recorded in the file *conf.h* in the form

#define Nclass number

where *class* is the device class name and *number* is the number of actual devices belonging to that class as defined in the device definition section. If there are no devices defined for a particular device class, no entry will be recorded in *conf.h*.

A device definition has the form

NAME is class [on typ] [-op function]* [addr= value]*

where *NAME* is a unique name corresponding to this device (or the name 'GENERIC'), *class* is a device class name defined in the first section of the configuration file, and *on typ* is a declared device type corresponding to the class. If there is only one type in a class the *on typ* may be omitted, otherwise it must be present. If *-op* and/or *addr=* fields are present, their values override those given in the device class and type declaration; otherwise the defaults are used.

Config uses each device definition to create an entry in *devtab* with each of the *devtab* fields filled in with the defaults or values given in the device definition line. Minor device numbers are computed consecutively among devices in each device class, starting with minor device number zero. The number of minor devices is the same as the number in the *Nclass* definition given above. The *NAME* field in a device definition is used to generate a line of the form

```
#define NAME          dvnum
```

in the file *conf.h*, where *dvnum* is the device number corresponding to the device. If the *NAME* field is given as *GENERIC*, then no such line is generated in *conf.h;* this is most commonly used when defining a number of identical devices belonging to the same device class and type.

C preprocessor statements such as

```
#include <dos.h>
```

may appear in the device definition section prior to the definitions themselves; these lines are added by *config* to the top of the file *conf.c* and are used, for example, in defining manifest constants used in the address fields of devices.

The *configuration constants* section is appended by *config* to the end of the file *conf.h* and is used principally to define configuration parameters such as the number of processes and semaphores and the version number printed at startup. Typically this section will contain information as illustrated in the following example:

```
/* Configuration and Size Constants */
#define MEMMARK
#define NPROC     30
#define NSEM      100

#define RTCLOCK

#define VERSION   "6pc (1-Dec-87)"
```

FILES

config/pcxconf, src/conf.h, src/conf.c

SEE ALSO

Intro(2)

Section 2: System Calls

The PC-Xinu operating system kernel consists of a set of run-time procedures to implement operating system services on an 8088 microcomputer system. The system supports multiple processes, I/O, synchronization based on counting semaphores, and preemptive scheduling. Each page in this section describes a system routine that can be called by a user process.

Each page describes one system call, giving the number and types of arguments that must be passed to the procedure under the heading "SYNOPSIS" (by giving their declaration in C syntax). The heading "SEE ALSO" suggests the names of other system calls that may be related to the described function. For example, the "SEE ALSO" entry for system call *wait* suggests that the programmer may want to look at the page for *signal* because both routines operate on semaphores.

In general, PC-Xinu blocks processes when requested services are not available. Unless that manual page suggests otherwise, the programmer should assume that the process requesting system services may be delayed until the request can be satisfied. For example, calling *read* may cause an arbitrary delay until data can be obtained from the device.

NAME

chprio – change the priority of a process

SYNOPSIS

int chprio(pid,newprio)
int pid;
int newprio;

DESCRIPTION

Chprio changes the scheduling priority of process *pid* to *newprio.* Priorities are positive integers. At any instant, the highest priority process that is ready will be running. A set of processes with equal priority is scheduled round-robin.

If the new priority is invalid, or the process id is invalid, *chprio* returns SYSERR. Otherwise, it returns the old process priority. It is forbidden to change the priority of the null process, which always remains zero.

SEE ALSO

create(2), getprio(2), resume(2)

BUGS

Because *chprio* changes priorities without rearranging processes on the ready list, it should only be used on waiting, sleeping, suspended, or current processes.

NAME

close – device independent close routine

SYNOPSIS

int close(dev)
int dev;

DESCRIPTION

Close will disconnect I/O from the device given by *dev*. It returns SYSERR if *dev* is incorrect, or is not opened for I/O. Otherwise, *close* returns OK.

Some tty devices like the console do not have to be opened and closed.

SEE ALSO

control(2), getc(2), open(2), putc(2), read(2), seek(2), write(2)

NAME

control – device independent control routine

SYNOPSIS

int control(dev, function, arg1, arg2)
int dev;
int function;
int arg1, arg2;

DESCRIPTION

Control is the mechanism used to send control information to devices and device drivers, or to interrogate their status. (Data normally flows through getc(2), putc(2), read(2), and write(2).)

Control returns SYSERR if *dev* is incorrect or if the function cannot be performed. The values returned otherwise are device dependent. For example, there is a control function for "tty" devices that returns the number of characters waiting in the input queue.

SEE ALSO

close(2), getc(2), open(2), putc(2), read(2), seek(2), write(2)

NAME

create – create a new process

SYNOPSIS

```
int create(caddr,ssize,prio,name,nargs[,argument]*)
char *caddr;
int ssize;
int prio;
char *name;
int nargs;
int argument;     /* actually, type machine word */
```

DESCRIPTION

Create creates a new process that will begin execution at location *caddr,* with a stack of *ssize* words, initial priority *prio,* and identifying name *name. Caddr* should be the address of a procedure or main program, If the creation is successful, the (nonnegative) process id of the new process is returned to the caller. The created process is left in the suspended state; it will not begin execution until started by a resume command. If the arguments are incorrect, or if there are no free process slots, the value SYSERR is returned. The new process has its own stack, but shares global data with other processes according to the scope rules of C. If the procedure attempts to return, its process will be terminated (see KILL(2)).

The process whose procedure name is *xmain* is created when PC-Xinu is initialized.

The caller can pass a variable number of arguments to the created process which are accessed through formal parameters. The integer *nargs* specifies how many argument values follow. *Nargs* values from the *arguments* list will be passed to the created process. The type and number of such arguments is not checked; each is treated as a single machine word. The user is cautioned against passing the address of any dynamically allocated datum to a process, because such objects may be deallocated from the creator's run-time stack even though the created process retains a pointer.

SEE ALSO

kill(2)

NAME

getc – device independent character input routine

SYNOPSIS

int getc(dev)
int dev;

DESCRIPTION

Getc will read the next character from the I/O device given by *dev*. It returns SYSERR if *dev* is incorrect. It returns the character read (widened to an integer) if successful.

SEE ALSO

close(2), control(2), open(2), putc(2), read(2), seek(2), write(2)

NAME

getdev – retrieve device number by device name

SYNOPSIS

int getdev(name)
char *name;

DESCRIPTION

Getdev will retrieve the number of a device given its device name. The device name is a string of length less than 10 which appears as the *dvnam* field in each *devtab* entry. Device numbers range from 0 to NDEVS-1. A common use of *getdev* is in *open* calls where the device is known only by its string name, as in

```
open(getdev("ds0"),"demo","r");
```

Getdev will return SYSERR if the name cannot be found or if the *name* string is empty.

Note that GENERIC devices cannot be retrieved by name.

SEE ALSO

control(2), getc(2), open(2), putc(2), read(2), seek(2), write(2)

NAME

getmem,getstk – get a block of main memory

SYNOPSIS

char *getmem(nbytes)
int nbytes;

char *getstk(nbytes)
int nbytes;

DESCRIPTION

Getmem and *getstk* are synonymous. *Getmem* rounds the number of bytes, *nbytes,* to an even-word multiple, and allocates a block of *nbytes* bytes of memory for the caller. *Getmem* returns the lowest word address in the allocated block. If less than *nbytes* bytes are available prior to the call, *getmem* returns SYSERR.

SEE ALSO

freemem(2), buffer(3)

BUGS

There is no way to protect memory, so any process may write into regions returned by *getmem.*

NAME

getpid – return the process id of the currently running process

SYNOPSIS

int getpid()

DESCRIPTION

Getpid returns the process id of the currently executing process. It is necessary to be able to identify one's self in order to perform some operations (e.g., change one's scheduling priority).

NAME

getprio – return the scheduling priority of a given process

SYNOPSIS

int getprio(pid)
int pid;

DESCRIPTION

Getprio returns the scheduling priority of process *pid.* If pid is invalid, *getprio* returns SYSERR.

NAME

init – device independent initialization routine

SYNOPSIS

int init(dev)
int dev;

DESCRIPTION

The operating system calls *init* once at system startup for every device configured into the device switch table. Argument *dev* gives the device descriptor of device to be initialized. *Init* returns SYSERR if *dev* is incorrect. Otherwise, it returns the value returned by the underlying device initialization routine.

Normally, user processes do not invoke *init* directly. However, because the exact semantics of device manipulation depend on underlying device-dependent routines, it may be possible or even necessary to do so for special devices.

SEE ALSO

close(2), control(2), getc(2), open(2), putc(2), seek(2), write(2)

NAME

kill – terminate a process

SYNOPSIS

int kill(pid)
int pid;

DESCRIPTION

Kill will stop process *pid* and remove it from the system, returning SYSERR if the process id is invalid, OK otherwise. *Kill* terminates a process immediately. If the process has been queued on a semaphore, it is removed from the queue and the semaphore count is incremented as if the process had never been there. Processes waiting to send a message to a full port disappear without affecting the port. If the process is waiting for I/O, the I/O is stopped (if possible).

One can kill a process in any state, including a suspended one. Once killed, a process cannot recover.

BUGS

At present there is no way to recover space allocated dynamically when a process terminates. However, *kill* does recover the stack space allocated to a process when it is created.

NAME

mark, unmarked – set and check initialization marks efficiently

SYNOPSIS

#include <mark.h>

int mark(mk)
MARKER mk;

int unmarked(mk)
MARKER mk;

DESCRIPTION

Mark sets *mk* to ‘‘initialized,’’ and records its location in the system. It returns 0 if the location is already marked, OK if the marking was successful, and SYSERR if there are too many marked locations.

Unmarked checks the contents and location of *mk* to see if it has been previously marked with the *mark* procedure. It returns OK if and only if *mk* has not been marked, 0 otherwise. The key is that marking works correctly after a reboot, no matter what was left in the marked locations when the system stopped.

Both *mark* and *unmarked* operate efficiently (in a few instructions) to determine whether a location has been marked. They are most useful for creating self-initializing procedures when the system will be restarted. Both the value in *mk* as well as its location are used to tell if it has been marked.

Memory marking can be eliminated from PC-Xinu by removing the definition of the symbol MEMMARK from the configuration file *pcxconf*. Self-initializing library routines may require manual initialization if MEMMARK is disabled (e.g., see BUFFER(3)).

BUGS

Mark does not verify that the location given lies in the static data area before marking it; to avoid having the system retain marks for locations on the stack after procedure exit, do not mark automatic variables.

NAME

open – device independent open routine

SYNOPSIS

open(dev)
int dev;

open(dev,name,mode)
int dev;
char *name;
char *mode;

open(dev,border,attr)
int dev;
char *border;
char *attr;

DESCRIPTION

Open will establish connection with the device given by *dev*. It returns SYSERR if *dev* is incorrect or if the specified device cannot be opened. Otherwise it returns the device number in *devtab* of the opened device.

The second form of *open* is used to open a disk file on a master device *dev* with filename *name*. The open mode is given in the string *mode* which can contain a combination of the characters **rwon.** Using 'r' will open the file for read access, while 'w' will open it for write access. The default is to open for both read and write access. Using 'o' (old) specifies that the file must already exist; in this case, *open* will return SYSERR if the file does not exist. Similarly, 'n' (new) specifies that the file must not exist, and *open* will return SYSERR if the file exists, otherwise it will create the file. It is an error to use both 'o' and 'n'. In the absence of either 'o' or 'n', *open* will create the file if it does not exist. If successful, *open* returns the device number of the device in *devtab* corresponding to the open file.

The third form of *open* will create a window on the master device *dev* with the given border and attribute strings. If successful, *open* returns the device number of the device in *devtab* corresponding to the open window.

The *border* string has the form "#c1,r1:c2,r2" where c1,r1 are the decimal coordinates (in column,row order) of the top left corner of the window, and c2,r2 are the coordinates of the bottom right corner. If the '#' character is omitted at the beginning of the *border* string, the window will be created without a border. The *attr* string has the form "fff/bbb". The 'fff' and 'bbb' strings are three-character color codes representing the foreground and background colors to be used with a color display. The color codes and corresponding colors are:

blk = black
blu = blue
grn = green
cyn = cyan
red = red
mag = magenta
yel = yellow
wht = white

The ‘fff’ or ‘bbb’ fields may be replaced by a single decimal digit in the range 0 to 7, which specifies the numeric code for the foreground or background color, respectively. For monochrome displays, the numeric code represents levels of gray with 0 for black and 7 for white. If either or both of the ‘fff’ or ‘bbb’ fields are missing, the appropriate defaults are taken. (If the ‘bbb’ field is missing, the ‘/’ may be omitted.)

The window color codes may be preceded by an optional blink specifier in the *attr* string; a ‘?’ blink specifier indicates that the foreground blinks, while a ‘∗’ specifies that it does not. Similarly, an optional intensity specifier may be given in the *attr* string; a ‘+’ specifies that the foreground is intensified, while a ‘-’ specifies that it is not.

The default window attributes are "∗-wht/blk" or, equvalently, "∗-7/0".

SEE ALSO

close(2), control(2), getc(2), putc(2), read(2), seek(2), write(2)

NAME

panic – abort processing due to severe error

SYNOPSIS

int panic(message)
char *message;

DESCRIPTION

Panic will print the character string *message* on the console, dump the machine registers and top few stack locations, and terminate PC-Xinu. It uses *kprintf* rather than *printf*, so it may be called anywhere in the kernel (e.g., from an interrupt routine that may be executed by the null process).

There are alternate entry points to *panic* that are invoked by divide by zero, illegal interrupts, or processor exceptions.

SEE ALSO

kprintf(3), printf(3)

NAME

pcount – return the number of messages currently waiting at a port

SYNOPSIS

int pcount(portid)
int portid;

DESCRIPTION

Pcount returns the message count associated with port *portid.*

A positive count p means that there are p messages available for processing. This count includes counts of messages explicitly in the port, the number of processes which are attempting to send a message to the port, and the number of processes which are attempting to send messages to the port but are blocked (because the queue is full). A negative count p means that there are p processes awaiting messages from the port. A zero count means that there are neither messages waiting nor processes waiting to consume messages.

SEE ALSO

pcreate(2), pdelete(2), preceive(2), preset(2), psend(2)

BUGS

In this version of PC-Xinu, SYSERR has the value -1 which corresponds to a legal port count.

NAME

pcreate – create a new port

SYNOPSIS

int pcreate(count)
int count;

DESCRIPTION

Pcreate creates a port with *count* locations for storing message pointers.

Pcreate returns an integer identifying the port if successful. If no more ports can be allocated, or if *count* is nonpositive, *pcreate* returns SYSERR.

Ports are manipulated with PSEND(2) and PRECEIVE(2). Receiving from a port returns a message that was previously sent to the port.

SEE ALSO

pcount(2), pdelete(2), preceive(2), preset(2), psend(2)

NAME

pdelete – delete a port

SYNOPSIS

int pdelete(portid, dispose) int portid; int (*dispose)();

DESCRIPTION

Pdelete deallocates port *portid.* The call returns SYSERR if *portid* is illegal or is not currently allocated.

The command has several effects, depending on the state of the port at the time the call is issued. If processes are waiting for messages from portid, they are made ready and return SYSERR to their caller. If messages exist in the port, they are disposed of by procedure *dispose*. If processes are waiting to place messages in the port, they are made ready and given SYSERR indications (just as if the port never existed). *Pdelete* performs the same function of clearing messages and processes from a port as preset(2) except that *pdelete* also deallocates the port.

SEE ALSO

pcount(2), pcreate(2), preceive(2), preset(2), psend(2)

NAME

preceive – get a message from a port

SYNOPSIS

preceive(portid)
int portid;

DESCRIPTION

Preceive retrieves the next message from the port *portid*, returning the message if successful, or SYSERR if *portid* is invalid. (The sender and receiver must agree on a convention for the meaning of the message.)

The calling process is blocked if there are no messages available (and reawakened as soon as a message arrives). The only ways to be released from a port queue are for some other process to send a message to the port with psend(2) or for some other process to delete or reset the port with pdelete(2) or preset(2).

SEE ALSO

pcount(2), pcreate(2), pdelete(2), preset(2), psend(2)

NAME

preset – reset a port

SYNOPSIS

int preset(portid, dispose)
int portid;
int (*dispose)();

DESCRIPTION

Preset flushes all messages from a port and releases all processes waiting to send or receive messages. *Preset* returns SYSERR if *portid* is not a valid port id.

Preset has several effects, depending on the state of the port at the time the call is issued. If processes are blocked waiting to receive messages from port *portid*, they are all made ready; each returns SYSERR to caller. If messages are in the port, they are disposed of by passing them to function *dispose*. If process are blocked waiting to send messages they are made ready; each returns SYSERR to its caller (as though the port never existed).

The effects of *preset* are the same as *pdelete*, followed by *pcreate*, except that the port is not deallocated. The maximum message count remains the same as it was.

BUGS

There is no way to change the maximum message count when the port is reset.

SEE ALSO

pcount(2), pcreate(2), pdelete(2), preceive(2), psend(2)

NAME

psend – send a message to a port

SYNOPSIS

int psend(portid, message)
int portid;
int message;

DESCRIPTION

Psend adds the integer *message* to the port *portid.* If successful, *psend* returns OK; it returns SYSERR if *portid* is invalid. Note that psend may return to the caller before the receiver has consumed the message.

If the port is full at the time of the call, the sending process will be blocked until space is available in the port for the message.

SEE ALSO

pcount(2), pcreate(2), pdelete(2), preceive(2), preset(2)

NAME

putc – device independent character output routine

SYNOPSIS

```
int putc(dev, ch)
int dev;
char    ch;
```

DESCRIPTION

Putc will write the character *ch* on the I/O device given by *dev*. It returns SYSERR if *dev* is incorrect, OK otherwise.

By convention, *printf* calls *putc* on device CONSOLE to write formatted output. Usually CONSOLE is device number zero.

SEE ALSO

close(2), control(2), getc(2), open(2), read(2), seek(2), write(2)

NAME

read – device independent input routine

SYNOPSIS

```
int read(dev, buffer, numchars)
int dev;
char *buffer;
int numchars;
```

DESCRIPTION

Read will read up to *numchars* bytes from the I/O device given by *dev*. It returns SYSERR if *dev* is incorrect, and returns the number of characters read if successful. The number of bytes actually returned depends on the device. For example, when reading from a device of type "tty", each read normally returns one line.

SEE ALSO

close(2), control(2), getc(2), open(2), putc(2), seek(2), write(2)

NAME

receive – receive a (one-word) message

SYNOPSIS

int receive()

DESCRIPTION

Receive returns the one-word message sent to a process using SEND(2). If no messages are waiting, *receive* blocks until one appears.

SEE ALSO

preceive(2), psend(2), receive(2)

NAME

resume – resume a suspended process

SYNOPSIS

int resume(pid)
int pid;

DESCRIPTION

Resume takes process *pid* out of hibernation and allows it to resume execution. If *pid* is invalid or process *pid* is not suspended, *resume* returns SYSERR; otherwise it returns the priority at which the process resumed execution. Only suspended processes may be resumed.

SEE ALSO

sleep(2), suspend(2), send(2), receive(2)

NAME

scount – return the count associated with a semaphore

SYNOPSIS

int scount(sem)
int sem;

DESCRIPTION

Scount returns the current count associated with semaphore *sem.* A count of negative p means that there are p processes waiting on the semaphore; a count of positive p means that at most p more calls to wait() can occur before a process will be blocked (assuming no intervening sends occur).

SEE ALSO

screate(2), sdelete(2), signal(2), sreset(2), wait(2)

BUGS

In this version of PC-Xinu, SYSERR has the value -1 which corresponds to a legal semaphore count.

NAME

screate – create a new semaphore

SYNOPSIS

int screate(count)
int count;

DESCRIPTION

Screate creates a counting semaphore and initializes it to *count. Screate* returns the integer identifier of the semaphore if successful, SYSERR if no more semaphores can be allocated.

Semaphores are manipulated with WAIT(2) and SIGNAL(2) to synchronize processes. Waiting causes the semaphore count to be decremented; decrementing a semaphore count past zero causes a process to be blocked. Signaling a semaphore increases its count, freeing a blocked process if one is waiting.

SEE ALSO

scount(2), sdelete(2), signal(2), sreset(2), wait(2)

NAME

sdelete – delete a semaphore

SYNOPSIS

int sdelete(sem)
int sem;

DESCRIPTION

Sdelete removes semaphore *sem* from the system and returns processes that were waiting for it to the ready state. The call returns SYSERR if *sem* is not a legal semaphore; it returns OK if the deletion was successful.

SEE ALSO

scount(2), screate(2), signal(2), sreset(2), wait(2)

NAME

seek – device independent position seeking routine

SYNOPSIS

int seek(dev, position)
int dev;
long position;

DESCRIPTION

Seek will position the device given by *dev* after the *position* character. It returns SYSERR if *dev* is incorrect, or if it is not possible to position *dev* as specified.

Seek cannot be used with devices connected to terminals.

Note that the position argument is declared *long* rather than *int.*

SEE ALSO

close(2), control(2), getc(2), open(2), putc(2), read(2), write(2)

NAME

send,sendf,sendn – send a (one-word) message to a process

SYNOPSIS

```
int send(pid, msg)
int pid;
int msg;

int sendf(pid, msg)
int pid;
int msg;

int sendn(pid, msg)
int pid;
int msg;
```

DESCRIPTION

In any of the three forms, *send* sends the one-word message *msg* to the process with id *pid*. A process may have at most one outstanding message that has not been received.

The form *send* returns SYSERR if *pid* is invalid, or if the process already has a message waiting that has not been received. Otherwise, it sends the message and returns OK. Processes are rescheduled following a *send.*

The form *sendf* differs from *send* only in that it forces the message *msg* to be sent to the process, even if it means destroying an existing message that has not been received.

The form *sendn* differs from *send* only in that it does not force a reschedule after the message has been sent.

SEE ALSO

preceive(2), psend(2), receive(2)

NAME

signal,signaln – signal a semaphore

SYNOPSIS

int signal(sem)
int signaln(sem,count)
int sem;
int count;

DESCRIPTION

In either form, *signal* signals semaphore *sem* and returns SYSERR if the semaphore does not exist, OK otherwise. The form *signal* increments the count of *sem* by 1 and frees the next process if any are waiting. The form *signaln* increments the semaphore by *count* and frees up to *count* processes if that many are waiting. Note that signaln(sem,x) is equivalent to executing signal(sem) x times.

SEE ALSO

scount(2), screate(2), sdelete(2), sreset(2), wait(2)

NAME

sleep,sleept – go to sleep for a specified time

SYNOPSIS

int sleep(secs)
int sleept(ticks)
int secs;
int ticks;

DESCRIPTION

In either form, *sleep* causes the current process to delay for a specified time and then resume. The form *sleep* expects the delay to be given in an integral number of seconds; it is most useful for longer delays. The resolution of the real-time clock used with PC-Xinu results in *sleep* calls to be in error by as much as 7 percent.

The form *sleept* expects the delay to be given in an integral number of ticks; it is most useful for short delays. Ticks occur at the rate of about 18.2 per second.

Both forms return SYSERR if the argument is negative or if the line time clock is not enabled on the processor. Otherwise they delay for the specified time and return OK.

Sleeping is not the same as hibernation (see SUSPEND(2)). In particular, sleeping processes cannot be awakened until they time out.

SEE ALSO

suspend(2)

BUGS

The maximum sleep is 32767 seconds (about 546 minutes, or 9.1 hours). Sleep guarantees a lower bound on delay, but since the system may delay processing of interrupts at times, sleep cannot guarantee an upper bound.

NAME

sreset – reset semaphore count

SYNOPSIS

int sreset(sem,count)
int sem;
int count;

DESCRIPTION

Sreset frees processes in the queue for semaphore *sem*, and resets its count to *count.* This corresponds to the operations of sdelete(sem) and sem=screate(count), except that it guarantees that the semaphore id *sem* does not change. *Sreset* returns SYSERR if *sem* is not a valid semaphore id. The current count in a semaphore does not affect resetting it.

SEE ALSO

scount(2), screate(2), sdelete(2), signal(2), wait(2)

NAME

suspend – suspend a process to keep it from executing

SYNOPSIS

int suspend(pid)
int pid;

DESCRIPTION

Suspend places process *pid* in a state of hibernation. If *pid* is illegal, or the process is not currently running and it is not on the ready list, *suspend* returns SYSERR. Otherwise *suspend* returns the priority of the suspended process. A process may suspend itself, in which case the call returns the priority at which the process is resumed. A process can put another into hibernation; a process can only put itself to sleep.

SEE ALSO

resume(2), sleep(2), send(2), receive(2)

NAME

wait – block and wait until semaphore signalled

SYNOPSIS

int wait(sem)
int sem;

DESCRIPTION

Wait decrements the count of semaphore *sem*, blocking the calling process if the count goes negative by enqueuing it in the queue for *sem*. The only ways to get free from a semaphore queue are for some other process to signal the semaphore, or for some other process to delete or reset the semaphore. *Wait* and *signal* are the two basic synchronization primitives in the system.

Wait returns SYSERR if *sem* is invalid. Otherwise, it returns OK once freed from the queue.

SEE ALSO

scount(2), screate(2), sdelete(2), signal(2), sreset(2)

NAME

write – write a sequence of characters from a buffer

SYNOPSIS

```
int write(dev, buff, count)
int dev;
char *buff;
int count;
```

DESCRIPTION

Write writes *count* characters to the I/O device given by *dev*, from sequential locations of the buffer, *buff*. *Write* returns SYSERR if *dev* or *count* is invalid, and the number of characters written for a successful write. *Write* normally returns when it is safe for the user to change the contents of the buffer. For some devices this means *write* will wait for I/O to complete before returning. On other devices, the data is copied into a kernel buffer and *write* returns while data is being transferred.

SEE ALSO

close(2), control(2), getc(2), open(2), putc(2), read(2), seek(2)

BUGS

Write may not have exclusive use of the I/O device, so output from other processes may be mixed in.

Section 3: Library Procedures

This section of the manual describes the procedures (functions) available to PC-Xinu processes from the PC-Xinu library. Additional library functions (such as string operations) are available from the C compiler run-time library; they are not documented here. Warning: most standard C run-time library functions will not work in PC-Xinu.

NAME

freebuf, getbuf, mkpool, poolinit – buffer pool routines

SYNOPSIS

```
int freebuf(buf);
char *buf;

char *getbuf(poolid);
int poolid;

int mkpool(bufsiz, numbufs)
int bufsiz, numbufs;

int poolinit()
```

DESCRIPTION

The routine *poolinit* initializes the entire buffer pool manager. It may be ignored as long as the MEMMARK option has been included in the PC-Xinu Configuration file *pcxconf*. Without MEMMARK, *poolinit* must be called once, before any other buffer manipulation routines.

Mkpool creates a pool of *numbufs* buffers, each of size *bufsiz*, and returns an integer identifying the pool. If no more pools can be created, or if the arguments are incorrect, *mkpool* returns SYSERR.

Once a pool has been created, *getbuf* obtains a free buffer from the pool given by *poolid*, and returns a pointer to the first word of the buffer. If all buffers in the specified pool are in use, the calling process will be blocked until a buffer becomes available. If the argument *poolid* does not specify a valid pool, getbuf returns SYSERR.

Freebuf returns a buffer to the pool from which it was allocated. *Freebuf* returns OK for normal completion, SYSERR if *buf* does not point to a valid buffer from a buffer pool.

BUGS

At present there is no way to reclaim space from buffer pools once they are no longer needed.

NAME

fgetc, getchar, kgetc, – get character from a device

SYNOPSIS

```
#include <io.h>

int fgetc(dev)
int dev;

int getchar()

int kgetc()
```

DESCRIPTION

Fgetc returns the next character from the named input *device*.

Getchar() is identical to *getc(CONSOLE)*.

Kgetc() is similar to *getchar()* , except that *kgetc* performs direct BIOS calls and can be used for system-level debugging.

Note that *fgetc* is exactly equivalent to *getc*.

SEE ALSO

getc(2), putc(2), gets(3), scanf(3),

DIAGNOSTICS

These functions return the integer constant **SYSERR** upon read error.

NAME

fputc, putchar, kputc – put character to a device

SYNOPSIS

```
#include <io.h>

int fputc(dev, c)
int dev;
char c;

putchar(c)
kputc(c)
char c;
```

DESCRIPTION

Fputc appends the character *c* to the named output *device*, and returns SYSERR if device is invalid; it is defined to be *putc*(2).

Putchar(c) is defined as *putc(CONSOLE, c). kputc(c)* is the same as *putchar(c)* , except that *kputc* performs direct BIOS calls and may be used for system-level debugging.

SEE ALSO

getc(3), puts(3), printf(3)

NAME

printf, fprintf, kprintf – formatted output conversion

SYNOPSIS

printf(format [**,** arg] ... **)**
char *format;

fprintf(dev, format [**,** arg] ... **)**
int dev;
char *format;

kprintf(format [**,** arg] ... **)**
char *format;

DESCRIPTION

Printf writes formatted output on device *CONSOLE*. *Fprintf* writes formatted output on the named output *device*. *Kprintf* writes formatted output to the console device using direct BIOS calls.

Each of these functions converts, formats, and prints its arguments after the format under control of the format argument. The format argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive *arg printf*.

Each conversion specification is introduced by the character **%**. Following the **%**, there may be, in the following order,

- an optional minus sign '–' which specifies *left adjustment* of the converted value in the indicated field;
- an optional digit string specifying a *field width;* if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- an optional period '**.**' which serves to separate the field width from the next digit string;
- an optional digit string specifying a *precision* which specifies the maximum number of characters to be printed from a string;
- the character **l** specifying that a following **d**, **o**, **x**, or **u** corresponds to a long integer *arg.* (A capitalized conversion code accomplishes the same thing.)
- a character which indicates the type of conversion to be applied.

A field width or precision may be '*' instead of a digit string. In this case an integer *arg* supplies the field width or precision.

The conversion characters and their meanings are

dox The integer *arg* is converted to decimal, octal, or hexadecimal notation respectively.

c The character *arg* is printed. Null characters are ignored.

s *Arg* is taken to be a string (character pointer), and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is 0 or missing all characters up to a null are printed.

u The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 through 65535 on the PC for normal integers and 0 through 4294967295 for long integers).

% Print a '%'; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by *putc*(2).

Examples

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

SEE ALSO

putc(2), kputc(3)

BUGS

Very wide fields (>128 characters) fail.