The XENIX® System V Development System

Release Notes

Version 2.3

The Santa Cruz Operation, Inc.

Document Number: X386/286-10-31-88-5.0/2.3

Part Number: 014-030-008

Processed Date: Wed Dec 21 02:07:26 PST 1988

XENIX System V 2.3
386/286 Development System
Release Notes

Release Notes
XENIX® 386/286 System V Development System
Version 2.3
For computers running the
XENIX-386, and XENIX-286 System V Operating System
October 31, 1988

1. Preface

These notes pertain to the SCO XENIX 386/286 System V Development System for computers running the SCO XENIX-386, and XENIX-286 System V Operating System. They contain notes on the software and documentation, and the procedure for installing the software.

We are always pleased to hear of a user's experience with our product, and recommendations of how it can be made even more useful. All written suggestions are given serious consideration.

2. Installation Notes

Note that you must have the XENIX System V Operating System installed on your computer in order to use the XENIX System V Development System.

To use the SCO XENIX System V release 2.3 Development System, you must have installed release 2.3 (or later) of the SCO XENIX System V Operating System.

Before you install the Development System, make sure that you have:

- The Development System diskettes.
- The Operating System "N" volume diskettes. Depending on your installation, you are prompted to insert one or more of these diskettes as part of your Development System installation procedure.

- Your Development System serial number and activation key.

To install the XENIX Development System, you must perform the following operations:

- First, log in as root (the "Super-User") and bring the system to system maintenance mode.
- Enter the command: **custom** and press RETURN.
- Select the Development System installation option.
- Install the Development System as prompted.

The XENIX System V Development System contains several packages that can be installed selectively using the **custom**(ADM) utility. For example, the DOS cross development environment (linker, libraries and include files) is a distinct package that you can either install or leave out of your system.

custom(ADM) can display a complete list and short description of the packages included in the Development System and the amount of disk space needed to install each package. You can use **custom**(ADM) at any time to install or remove all or part of the Development System. Refer to the **custom**(ADM) manual page for instructions on using **custom**(ADM).

During the installation, you are asked to choose a default terminal description library for use with the **curses**(S) screen handling package. You can choose **terminfo**, **termcap**, or you can choose to have no default package, by selecting neither. The **curses** include file (*/usr/include/curses.h*) will conditionally include the correct information that will enable it to be used with **terminfo** or **termcap** depending on whether M_TERMINFO or M_TERMCAP is defined. The choice you make at installation time determines which of these packages is the default for your system. You can always override the default by explicitly defining either M_TERMINFO or M_TERMCAP either in your program before the statement that

includes *curses.h* in the source code, or by using the **-D** directive on the C compiler command line. If you do not select a default, you must **#define** either M_TERMINFO or M_TERMCAP whenever you compile a source module that includes *curses.h*.

Towards the end of the installation procedure you are prompted to insert one or more of the "N" volumes. These diskettes are not part of the Development System distribution. (Although **custom**(ADM) may refer to them as "Development System" diskettes.) They are volumes from the Operating System distribution and certain Operating System dependent files and utilities must be extracted from them when you install the Development System.

2.1 Packages In The Development System

The 386 Development System consists of the following packages:

Development System Packages

Name	Description	Size
ALL	Entire Development System set	13694
SOFT	Basic software development tools	8254
LEX	Generates programs for lexical analysis	114
YACC	Yet another compiler-compiler	106
CREF	Cross reference programs	466
CFLOW	Generates C flow graphs	114
LINT	Syntax and usage check files and tools	402
SMALL	Small Model Library routines	442
MEDIUM	Medium Model Library routines	460
COMPACT	Compact Model Library routines	472
LARGE	Large model Library routines	490
SCCS	Source code control system	692
DOSDEV	DOS cross development libraries and utilities	1460

HELP	Help utility and related files	166

Note that the SMALL, MEDIUM, and LARGE packages in the 386 development system allow development of programs to run on 8086 and 80286 systems. 80386 systems do not use memory models.

Once you have installed the 80386 Development System, please check the file */etc/default/cc* to confirm that the default settings for the compiler are those you desire.

The 286 Development System consists of the following packages:

Development System Packages

Name	Description	Size
ALL	Entire Development System set	9466
SOFT	Basic software development tools	3652
LEX	Generates programs for lexical analysis	94
YACC	Yet another compiler-compiler	88
CREF	Cross reference programs	368
CFLOW	Generates C flow graphs	86
LINT	Syntax and usage check files and tools	274
SMALL	Small Model Library routines	612
MEDIUM	Medium Model Library routines	768
COMPACT	Compact Model Library routines	544
LARGE	Large model library routines	862
SCCS	Source code control system	558
DOSDEV	DOS cross development libraries and utilities	1214
HELP	On-line help utility	172

Once you have installed the 80286 Development System, please check the file */etc/default/cc* to confirm that the default settings for the compiler are those you desire.

3. Software Notes

3.1 Include Files

Some include files shipped with releases 2.3.0 and 2.3.1 of XENIX were missing or incorrect. These files have been re-shipped with this distribution of the Development System. These files can be found in */usr/include/sys_2.3*. If you have release 2.3.0 or 2.3.1 of the XENIX Operating System, you can move these include files to */usr/include/sys* to correct any difficulties.

3.2 MINDOUBLE

The XENIX C compiler does not compute the value of the constant MINDOUBLE correctly. It always evaluates to 0 when used in expressions. This constant is defined in */usr/include/values.h*.

3.3 286 Floating Point Emulation

286 floating point emulation software in 286 Operating Systems prior to 2.3 does not correctly execute floating point comparisons between the top of the floating point stack and memory. When you develop applications using the **-dos**, or the **-M0**, **-M1**, or **-M2** flags, you should use the **-Go** option to **cc**(CP) to instruct the Compiler to allow for this. Note that the **-compat** flag implies the **-Go** option automatically.

When using masm, special care needs to be taken not to make floating comparisons with memory operands. To execute correctly on previous 286 floating point emulators, only comparisons of different stack elements should be made. Note that pushing a memory operand onto the stack causes the source/destination relation between the two operands to be reversed in a subsequent comparison operation. As an example:

fcom *memory_address*
jg *label*

should be written as:

fld *memory_address*
fcomp *ST(1)*
jl *label*

3.4 Substituting Parameters in Quoted Strings

There is a commonly used programming technique that must be slightly altered to operate correctly when compiling 286 binaries. The proposed ANSI standard for C does not allow the substitution of macro formal parameters within quoted strings. However, many existing C compilers, including the XENIX 386 C compiler, do allow this. Consider the following macro definition and use:

```
#define CTRL(c)          ('c' & 0x1f)

putchar(CTRL(g));
```

The above statement operates correctly on 386 machines. The existing 80386 compiler expands this to:

```
putchar(('g' & 0x1f));
```

But according to the ANSI standard it must be:

```
putchar(('c' & 0x1f));
```

The 286 compiler follows the ANSI standard strictly, therefore you must use this substitution as follows:

```
#define CTRL(c)          (c & 0x1f)

putchar(CTRL('g'));
```

Then the compiler expands this to:

```
putchar(('g' & 0x1f));
```

The **-Me** option to the 386 C compiler allows the non-ANSI functionality described above, but displays a warning that a non-standard extension has been used.

3.5 Transforming Formal Parameters to Strings

The ANSI standard defines a mechanism for transforming macro formal parameters into quoted strings (the stringising operator, ''#'') and an operator for pasting strings, ''##''. These have been added to both the 286 and 386 preprocessors. A slight restriction on the 386 is that white space is not allowed between the stringising operator and the formal parameter it operates on.

3.6 setvbuf(S)

The order of the parameters to **setvbuf**(S) has been changed to conform with the SVID. Prior to release 2.3, the order of the parameters was:

```
int setvbuf (stream, type, buf, size)
```

In release 2.3, the order of the parameters is:

```
int setvbuf (stream, buf, type, size)
```

The files */lib/compat*[LMCS]*setvbuf.o* can be used for backwards compatibility.

3.7 Terminfo Curses

When you use terminfo curses in a 286 binary, you should compile with the **-F** option set to 2000 or greater to increase the fixed stack size. For small model programs, compile with the **-i** flag.

3.8 Tokens Following #else and #endif

The C compiler issues a warning if it finds anything other than blank space or comments following an **#else** or **#endif** preprocessor directive. Thus the first example below would produce a warning but the second would not.

```
#endif  M_I386
#endif  /* M_I386 */
```

These warnings are only produced if the warning level is set to 2 or greater on the 386 compiler, but are produced at the default level of 1 on the 286 compiler.

3.9 COFF Support

The SCO XENIX System V Development System supports the transparent execution of COFF binaries. Certain utilities also operate transparently on COFF binaries. These include ar(CP), adb(CP), directory(S), ld(CP), nm(CP), prof(CP), nlist(S), and xlist(S). Where there is no direct symbol matching, utilities make the closest possible translation of COFF symbols.

3.10 Include Files and Utilities

The machine dependent Development System *include* files and utilities are included on the XENIX version 2.1.3 (or later) Operating System N Volumes. The **terminfo** database is part of the XENIX Operating System extended utilities not available prior to release 2.2.

3.11 crypt(C) and crypt(S) Libraries

The **crypt**(C) utility and **crypt**(S) libraries are not included in this release. If you are a United States resident, you can request a copy of **crypt** by calling SCO support.

3.12 cxref(CP)

cxref(CP) does not correctly handle function prototypes. It's syntax analysis is based on an older version of the compiler. To work around this problem, make function prototypes subject to a conditional compilation definition. For example:

```
#ifndef NO_PROTOTYPES
int func( char *, long );
#endif
```

Then use the flag:

-DNO_PROTOTYPES

when using **cxref**.

3.13 cc(CP) Notes

3.13.1 Register Variables

Attempts to compile certain complex expressions that involve pointer arithmetic and register variables may fail with an internal compiler error.

Usually, removing the "register" storage class specifier from the declarations avoids this problem, but occasionally you must simplify the expression by splitting it into several, less complex, expressions.

Note that only objects of type *int*, *short*, or *char* and the unsigned versions of these types are candidates for register storage class in large model programs. Unless such an item is accessed very frequently, making it an "auto" rather than a "register" item probably does not have much impact on the program's performance.

If you do not wish to alter your source code, add this flag to your **cc** command line to remove all register declarations from a program:

-Dregister=auto

3.13.2 -Zi Option

The **-Zi** option to **cc**(CP) cannot be used if function prototypes with a variable number of arguments are present. If you have a declaration such as:

```
int vargfunc( char *, ... );
```

in your code, then the **-Zi** option cannot be used. If you wish to use **-Zi** when prototypes with ellipses (...) are present, the prototype should be conditionally declared as follows:

```
#ifndef DEBUG
int vargfunc( char *, ... );
#endif
```

3.13.3 Variable Declarations

The error: *Segmentation Violation: core dumped* may occur if you declare too many variables within a single declaration statement. For example, the following text may cause the compiler to dump core:

```
char *
        x1,     /* comment */
        x2,     /* more comment */
        .
        .
        .
        x934;   /* more comment */
```

Twenty five variables in a declaration is a safe maximum. Break longer declarations into two or more statements to avoid this possible problem.

3.14 sdb(CP)

Programs that are to be debugged with **sdb** must be compiled and linked with the following options:

cc -Zi or **cc -g**

ld -g

The **-g** option should only be used when invoking **ld** directly.

The following features of **sdb** that are described in the documentation are not supported in this release:

- Pattern matching for function and variable names. For example,

  x*/

  should display the values of all variables with names beginning with "x".

- The "." command to redisplay the value of the last variable typed.

- The command:

  * <*file*

  to read **sdb** commands in from a file works correctly only when there is no space placed between the "<" character and the file name.

3.15 ld

The **-r** option of **ld**(CP) does not work correctly for object files that contain the additional symbol table information used by the symbolic debugger. If you use the **-r** option of **ld** to partially link together several object modules that contain symbolic debugging information, the code, data, and relocation information in the resulting object module will be correct, but the symbolic debugging information will be unusable.

3.16 irand48() and krand48()

The functions **irand48**() and **krand48**() are not supported in this release.

3.17 asx(CP)

The pre-cmerge assembler is included with this release for those users who have programs that require it. It is called */bin/asx* and is documented on the manual page **asx**(CP).

3.18 stdio.h

The buffer size used by the standard I/O library is now 1024 bytes. This change is reflected in the standard I/O header file */usr/include/stdio.h* where **BUFSIZ** is defined as 1024 bytes.

This change does not affect existing XENIX executable binaries. However, it is important that any object modules that use standard I/O functions and that were compiled using the old *stdio.h* with a **BUFSIZ** of 512 bytes should be recompiled before they are linked with the new libraries.

An alternative is to include the file */lib/compat/*[SML]*setbuf.o* on the command line when you link your program. This enables the new libraries to work correctly with object modules that assume a **BUFSIZ** of 512 bytes.

You should take particular care when you install products on XENIX that are in the form of linkable objects rather than executable binaries. It is generally necessary to include */lib/compat/*[SML]*setbuf.o* in the link.

For example, if the installation procedure for a product includes a link command such as:

```
cc -Mm -i -o prog proglib.a progsub.o -ltermlib
```

You should edit it to read:

```
cc -M0m -i -o prog proglib.a progsub.o /lib/compat/Msetbuf.o -ltermlib
```

Note that it is safe to include */lib/compat/*[SML]*setbuf.o*, even if the application was compiled with a standard I/O **BUFSIZ** of 1024 bytes. The only consequence is that your buffer size is reduced to 512 and the standard I/O package is slightly less efficient.

386 Development System users should also note that since their default code generation mode is **-M3e**, they need to explicitly specify **-M0** or **-M2** when they link 8086 or 80286 code.

3.19 types.h

Some of the C language *.h* files require that *<sys/types.h>* be included first. An error message generally occurs when a type is used in an #include file but not declared. Often, the problem is corrected by including *<sys/types.h>* earlier in the program.

3.20 malloc Issues

There are two versions of **malloc** distributed with the XENIX Development System. The standard **malloc** is contained in the file */lib/libc.a*. There is an alternate **malloc** in */lib/libmalloc.a*. Both are documented under the **malloc**(CP) manual page.

If your program uses many **malloc** and **free** calls, running the program under the XENIX 386 Operating System may cause an excessive page swapping problem as **malloc** must search the entire list of allocated and free blocks. If you are using **malloc** and **free** calls extensively, it is suggested that you use the alternate **malloc** found in */lib/libmalloc.a*. To use */lib/libmalloc.a*, compile your program with the **-lmalloc** flag on your **cc**(CP) command line. Note that the alternate **malloc** found in */lib/libmalloc.a* does not include a large model version.

The alternate **malloc** maintains a separate list of free blocks and may be faster, but data in any allocated block is immediately overwritten when the block is declared free. The standard **malloc** preserves data between consecutive **free** and **malloc** calls. Some programs rely on this functionality of the standard **malloc**.

3.20.1 mallinfo()

Note that you must call **malloc**() at least once before calling **mallinfo**(). Otherwise, calling **mallinfo**() may result in unpredictable behavior.

4. Operating System Specific Software Notes

4.1 80386 Operating System

4.1.1 cc(CP) Defaults

The default code generation model for the 80386 is **-M3e**. (80386 with non-ANSI extensions enabled.) This default can be changed by editing */etc/default/cc*. See the **cc**(CP) manual page for details.

4.1.2 String Constants

The 386 C compiler has been extended to allow 4K bytes in string constants. The compiler fails ungracefully if this limit is exceeded. 386 C compiler:

4.1.3 Variable Assignments

Assignments of the following form generate incorrect code when compiled with optimization enabled:

```
int i;
register  char *buf;

buf[ i + i ] = buf[ i ];
buf[ i ] = buf[ i + i ];
```

Note that if "buf" is not a register variable, or if the index expression "i + i" contains any term other than "i", such as a constant or another variable, correct code is generated. Thus:

```
buf[ i + i + 1 ] = buf[ i ];
buf[ i + n ]    = buf[ i ];
```

both work correctly.

4.1.4 brkctl(S) Library

XENIX 386 does not support 386 processes which have more than one data segment. For compatibility, a special library (*/lib/386/Slibbrkctl.a*) has been provided that maps **brkctl**() functions into calls to **sbrk**(S). This provides for the most common uses of **brkctl**(S), such as the allocation of additional memory.

Programs which rely on allocating multiple data segments and manipulating the sizes of those segments will need to be altered to work under XENIX 386.

The following describes the functionality implemented by the 386 *libbrkctl.a*. Note in particular that the 386 **brkctl**(S) returns a near pointer and that the third argument is also a near pointer. If you do not wish to alter your source code when compiling for the 386 you should include **-Dfar[=string]** on the **cc**(CP) command line. This will cause the preprocessor to remove all "far" keywords from your code.

```
#include <sys/brk.h>
char *brkctl(cmd, inc, ptr)
int cmd;
long inc;
char *ptr;
```

When called from 386 processes **brkctl**() does not cause the allocation or de-allocation of far data segments, it can only be used to increase or decrease the memory allocation in the single data segment available to 386 processes.

Any of the commands **BR_IMPSEG**, **BR_ARGSEG** or **BR_NEWSEG** can be used as they all have the same effect.

The *ptr* argument in the above program example is ignored.

It is unusual to allow the use of **BR_NEWSEG** with 386 processes since it is not possible to allocate additional data segments. However, since **BR_NEWSEG** is commonly used in small and middle model 86/286 processes to allocate additional memory it was decided to allow **BR_NEWSEG** but to make its functionality the same as **BR_IMPSEG**.

In order to link a 386 binary with this version of **brkctl**() you should specify **-lbrkctl** on the **cc**(CP) command line.

4.1.5 masm

In addition to supporting the 386 instruction set, the 386 version of **masm** shipped with the 386 Development System has some minor differences in source code syntax from previous versions. For compatibility, the 86/286 **masm** has been included in this distribution as */bin/masm86*.

4.1.6 Memory Models

The only memory model supported for 386 code is ''small'' model. Small model has two 32 bit segments; one for code and one for data. Small, middle, large, and huge models are supported for 8086/286 code.

4.1.7 nm(CP)

The **-n** option of **nm**(CP) generates an error with long name lists. To sort a long name list, use the command line:

nm *executable* | sort

instead of the **-n** option.

4.1.8 Stack Size

80386 programs have a variable sized stack that is expanded by the kernel as needed. 8086/286 programs run under the 386 Operating System have a fixed size stack. The default stack size for 8086/286 binaries is 4 Kilobytes unless the **-F** option of the linker was used. Once compiled, you can change the stack size of an 8086/286 program using the **-F** option to **fixhdr**(C).

Preprocessor Names

Symbols which are used in **#define, #ifdef** and other preprocessor commands must now conform with the same rules as those for C identifiers. They must begin with an alphabetic character or an underscore and may only contain alphanumeric characters and the underscore character.

Previous versions of the C compiler allowed other non-alphanumeric characters such as "." in preprocessor symbols.

Stricter Checking of Storage Class in Declarations

The ANSI standard requires that declarations and definitions of functions and variables must have matching storage classes as well as matching types. Typically, this causes problems in code when a function is first used (and implicitly declares it as an "extern int") and is later defined as "static int." ANSI requires that the first use of the function be preceded by an explicit declaration. Thus:

```
static int foo(); /*required by ANSI C */
Program text
        foo();
static foo()
```

The **-Me** option disables this strict checking of storage class.

4.1.9 sccs(CP) Line Size

There is a limit of 508 characters on any physical line of any file placed under **sccs**. Writing a changed file with a line containing more than 508 characters results in corruption of the file.

4.2 80286 Operating System

4.2.1 cc(CP) Defaults

The default code generation model is **-M0** (8086 small model). This default can be changed by editing */etc/default/cc*. See the **cc**(CP) manual page for details.

4.2.2 brkctl(S)

brkctl(S) is a XENIX specific system call that can be used by 86/286 processes to allocate memory in far data segments. (see **brkctl**(S).)

The full functionality of **brkctl**() is only available to 8086/286 binaries running under the XENIX 286 or XENIX 386 Operating Systems.

XENIX 86 does not support the allocation of additional data segments, therefore when **brkctl**() requests that require this are made by an 8086 binary running under XENIX 86 they will fail. The **-compat** option of the C compiler can be used to link 8086 binaries with a special version of **brkctl**() (in */lib/*[SML]*libbrkctl.a*) that satisfies requests for additional data segments under XENIX 86 by allocating shared memory segments.

4.2.3 tty.h

In the file */usr/include/sys/tty.h*, **t_proc** is an:

```
int (far *t_proc)()
```

When including this file, be sure to enable **near** and **far** keywords by using the **-Me** flag to **cc**(CP).

4.2.4 ld(CP)

The **-S** option of **ld**(CP) only accepts values up to 8192. Any argument greater than 8192 returns the error message "segment limit set too high".

4.2.5 Stack Size Limitations

The default stack for programs running under the XENIX 286 Operating System is a fixed size stack of 1000 hexadecimal bytes. Use the **-F** flag to specify a different stack size. Variable stack size is not supported by the XENIX 286 Operating System. Note that you can change the stack size on a compiled program with the **fixhdr**(C) command.

4.2.6 Floating-Point Exceptions

For compatibility reasons, floating-point exceptions (like dividing by zero) are masked within the **libc** libraries for 286 binaries. 286 binaries should check the floating point status word for evidence of masked exceptions.

5. Differences Between 286 and 386 Code Generation

In general there will be few problems in recompiling existing C programs to run on the 80386. However, the following differences between the 8086/80286 and the 80386 should be noted.

5.0.1 Size of Integers

386 integers are 32 bits whereas 86/286 integers are 16 bits. This may cause problems in programs that exchange binary data with other programs or programs that have to read binary data from existing data files. Note that the signed and unsigned short data types are 16 bits on both 86/286 and 386 processors.

5.0.2 Size of Pointers

386 small model code and data pointers are 32 bits. On the 8086/286, the size of pointers depends on the memory model.

5.0.3 Assembly language interface

The 386 assembly language interface is very similar to the 86/286 assembly language interface, but note that 386 C code may use up to 3 register variables (**esi, edi, ebx**) whereas 86/286 code used at most 2 variables (**si, di**). It is essential that 386 assembly language routines preserve the contents of **ebx** as well as **esi** and **edi.**

5.0.4 Zp2 and Zp4 structure alignment

There are different rules for calculating the size of structures and alignment of structure members on the 386 and the 86/286. These differences are described in detail in Chapter 2 of the *C User's Guide.*

The **#pragma pack()** directive is supported by the 386 compiler. This directive allows you to override the default structure packing rules within a C source file.

You may specify **pack(1)**, **pack(2)**, or **pack(4)** after the **#pragma** directive. This has the same effect as placing one of the **-Zp1**, **-Zp2**, or **-Zp4** command line arguments in your C compiler

command. **pack()** with no arguments returns structure packing to whatever was explicitly specified on the **cc** command line or the default (4) if nothing was specified.

This directive is useful for insuring that structures have the same alignment regardless of whether they are part of 286 or 386 binaries. Note that this directive is supported by both the 286 and 386 C compilers.

Note that the 80386 C libraries are compiled with the **-Zp4** flag. It is important that you compile any programs that pass information to and from library functions and system calls in structures with the **-Zp4** flag. When you compile a program with **-Zp4** set, you should use the **#pragma pack(2)** construct to select two byte alignment for any structures that need two byte alignment.

6. Documentation Errata

6.1 Device Driver Writer's Guide

There is an error on pages 6-1 to 6-2 in the examples for **db_alloc()** and **db_free()**. In the examples for each of these functions, you see the following line:

```
struct devbuf tb[2];
```

This line should read:

```
struct devbuf tb[2], *ptb = tb;
```

Then, in the **db_alloc()** example, you see:

```
if( db_alloc( &tb, 2 ) == 0 ) {
```

This line should read:

```
if( db_alloc( &ptb, 2 ) == 0 ) {
```

In the **db_free()** example, you see:

```
db_free( &tb, 2 );
```

This line should read:

```
db_free( &ptb, 2 );
```

6.2 cc(CP)

The **-n** flag to **cc**(CP) is documented as having the same function as the **-i** flag. However, the **-n** flag instructs the linker to produce an impure non-split i/d binary.

6.3 dosld(CP)

The dosld(cp) man page is incorrect in stating that the **-C** flag makes this utility case insensitive. This utility is case insensitive by default. The **-C** flag makes this utility case sensitive.

6.4 dirent(F)

The **dirent**(F) manual page is included with these Release Notes. Please add this page to the (F) manual page section of your XENIX *User's Reference*. This page describes the file formats related to directory searching with the **getdents**(S) system call.

6.5 sigset(S)

The **sigset**(S) manual page is also included with these Release Notes. Please add this page to the (S) manual page section of your XENIX *Programmer's Reference*. This page describes various signal maintenance routines.

6.6 curses(S)

The following functions should be documented in the **curses(S)** manual page:

addkey()
keypad()
dmpwin()
wattron()
wattroff()
wattrset()

These functions and other **curses** features are currently

documented in Chapter 7 of the XENIX *C Library Guide*.

Name

dirent - file system independent directory entry

Syntax

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

Description

Different file system types may have different directory entries. The *dirent* structure defines a file system independent directory entry, which contains information common to directory entries in different file system types. A set of these structures is returned by the *getdents*(S) system call.

The *dirent* structure is defined below.

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

The *d_ino* is a number which is unique for each file in the file system. The field *d_off* is the offset of that directory entry in the actual file system directory. The field *d_name* is the beginning of the character array giving the name of the directory entry. This name is null terminated and may have at most MAXNAMLEN characters. This results in file system independent directory entries being variable length entities. The value of *d_reclen* is the record length of this entry. This length is defined to be the number of bytes between the current entry and the next one, so that it will always result in the next entry being on a long boundary.

Files

/usr/include/sys/dirent.h

See Also

getdents(S).

Name

sigset, sighold, sigrelse, sigignore, sigpause - signal management routines.

Syntax

#include <signal.h>

void (*sigset (sig, func))()
int sig;
void (*func)( );

int sighold (sig)
int sig;

int sigrelse (sig)
int sig;

int sigignore (sig)
int sig;

int sigpause (sig)
int sig;

Description

These functions provide signal management for application processes. The *sigset* system call specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

Sig can be assigned any one of the following values except SIGKILL. Machine- or implementation-dependent signals are not included (see **Notes** below). Each value of **sig** is a macro, defined in **<signal.h>**, that expands to an integer constant expression.

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction (not held when caught)
SIGTRAP*	trace trap (not held when caught)
SIGABRT*	abort
SIGFPE*	floating point exception
SIGKILL	kill (cannot be caught or ignored)
SIGSYS*	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2
SIGCLD	death of a child (see **WARNING** below)

SIGPWR power fail (see **WARNING** below)
SIGPOLL selectable event pending (see **NOTES** below)

See below under SIG_DFL regarding asterisks (*) in the above list.

The following values for the system-defined actions of *func* are also defined in *<signal.h>*. Each is a macro that expands to a constant expression of type pointer to function returning *void* and has a unique value that matches no declarable function.

SIG_DFL—default system action
Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(S). In addition a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(S)]

a file owner ID that is the same as the effective user ID of the receiving process

a file group ID that is the same as the effective group ID of the receiving process.

SIG_IGN—ignore signal
Any pending signal *sig* is discarded and the system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD—hold signal
The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

Otherwise, *func* must be a pointer to a function, the signal-catching handler, that is to be called when signal *sig* occurs. In this case, *sigset* specifies that the process will call this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here.

When a signal occurs, the signal number *sig* will be passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action will be set to SIG_HOLD. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

In general, upon return from the signal-catching handler, the receiving process will resume execution at the point it was interrupted. However, when a signal is caught during a *read*(S), a *write*(S), an *open*(S), or an *ioctl* (S) system call during a **sigpause** system call, or during a *wait*(S) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler will be executed. Then the interrupted system call may return a -1 to the calling process with *errno* set to EINTR.

sighold and *sigrelse* are used to establish critical regions of code. *sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

sigignore sets the action for signal *sig* to SIG_IGN (see above).

sigpause suspends the calling process until it receives a signal, the same as *pause*(S). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, then call *sigpause* to wait for the signal. **sigset** will fail if one or more of the following is true:

[EINVAL] *Sig* is an illegal signal number (including **SIGKILL**) or the default handling of *sig* cannot be changed.

[EINTR] A signal was caught during the system call *sigpause*.

See Also

kill(S), pause(S), signal(S), wait(S), setjmp(S).

Diagnostics

Upon successful completion, *sigset* returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is

defined in *<signal.h>*.

For the other functions, upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

Notes

SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see **intro**(S)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG **ioctl**(S) call. Otherwise, the process will never receive **SIGPOLL**.

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. The action for the signal **SIGKILL** cannot be changed from the default system action.

Specific implementations may have other implementation-defined signals. Also, additional implementation-defined arguments may be passed to the signal-catching handler for hardware-generated signals. For certain hardware-generated signals, it may not be possible to resume execution at the point of interruption.

The signal type **SIGSEGV** is reserved for the condition that occurs on an invalid access to a data object. If an implementation can detect this condition, this signal type should be used.

The other signal management functions, *signal*(S) and *pause*(S), should not be used in conjunction with these routines for a particular signal type.

Warnings

Two signals that behave differently from the signals described above exist in this release of the system:

SIGCLD	death of a child (reset when caught)
SIGPWR	power fail (not reset when caught)

For these signals, *func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit* (S)].

s**function address** - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

The **SIGCLD** affects two other system calls [*wait* (S), and *exit* (S)] in the following ways:

wait If the *func* value of **SIGCLD** is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to SIG_IGN, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

10-31-88

014-030-008

P.O.22858
PAT 050