The XENIX® System V Development System

Release Notes

Version 2.2

The Santa Cruz Operation, Inc.

This document was typeset with an IMAGEN® 8/300 Laser Printer.

IMAGEN is a registered trademark of IMAGEN Corporation.
XENIX is a registered trademark of Microsoft Corporation.

Document Number: X86/286/386-6-29-87-4.0/2.2

Processed Date: Sun Jun 28 18:43:51 PDT 1987

XENIX System V 2.2 86/286/386 Development System Release Notes

Release Notes
XENIX 86/286/386 Development System
Release 2.2
June 29 1987

1. Preface

These notes pertain to the XENIX System V Development System Release 2.2 for personal computers. 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.

For the 86/286 Development System, you must have installed release 2.1.3 (or later) of the XENIX Operating System.

For the 386 Development System you must have installed release 2.2.1 (or later) of the XENIX 386 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 single-user operation.
- 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**(C) utility. For example, the DOS development environment (linker, libraries and include files) is distinct package that you can either install, or leave out of your system.

custom(C) can also 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 also use **custom**(C) at any time to install or remove all or part of the Development System. Refer to the **custom**(C) manual page for instructions on using **custom**(C).

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 decline to choose a default 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 is the default for your system. You can always override the deafult 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**(C) 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 Development System for the XENIX 8086 Operating System consists of the following packages:

Development System Packages

ALL	Entire Development System set
SOFT	Basic software development tools
LEX	Generates programs for lexical analysis
YACC	Yet another compiler-compiler
CREF	Cross reference programs
CFLOW	Generates C flow graphs
LINT	Syntax and usage check files and tools
MEDIUM	Medium Model Library routines
LARGE	Large model library routines
SCCS	Source code control system
DOSDEV	DOS cross development libraries and utilities

The Development System for the XENIX 80286 Operating System consists of the following packages:

Development System Packages

ALL	Entire Development System set
SOFT	Basic software development tools
LEX	Generates programs for lexical analysis
YACC	Yet another compiler-compiler
CREF	Cross reference programs
CFLOW	Generates C flow graphs
LINT	Syntax and usage check files and tools
MEDIUM	Medium Model Library routines

LARGE	Large model library routines
CCL	Large model compiler passes
SCCS	Sourcecode control system
DOSDEV	DOS cross development libraries and utilities

The Development System for the XENIX 80386 Operating System consists of the following packages:

Development System Packages

ALL	Entire Development System set
SOFT	Basic software development tools
LEX	Generates programs for lexical analysis
YACC	Yet another compiler-compiler
CREF	Cross reference programs
CFLOW	Generates C flow graphs
LINT	Syntax and usage check files and tools
SMALL	Small Model 8086 Library routines
MEDIUM	Medium Model 8086 Library routines
LARGE	Large model 8086 Library routines
SCCS	Source code control system
DOSDEV	DOS cross development libraries and utilities

3. Software Notes

This release includes the Microsoft C compiler with new passes, a new linker, and a source code debugger for C programs (**sdb**).

3.1 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.

3.2 crypt(C) and crypt Libraries

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

3.3 cc(CP) Notes

3.3.1 Large Model Code Generation

This release supports large model 8086 or 80286 program generation for XENIX and DOS. Large model programs developed on 8086 machines under the XENIX 86 Operating System will only run on 80286 or 80386 machines under the XENIX 286 or XENIX 386 Operating System.

3.3.2 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

3.3.3 Huge Model Code Generation

Expressions similar to the following do not compile properly in huge model:

```
char *p, *foo;
if (--p < foo)
```

Use this type of construction instead:

```
--p;
if (p < foo)
```

3.3.4 Variable Declarations

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

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

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

3.4 sdb (CP)

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.
- Restoring non-blocking reads. If your program is performing a non-blocking read when it reaches an **sdb** breakpoint, **sdb** does not return to non-blocking reads and must be terminated with EOF.
- The **e** command to display current function name. The **e** command does not return the current function name as described.

3.5 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 unuseable.

3.6 cxref(CP)

cxref(CP) is not supported in this release.

3.7 irand48() and krand48()

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

3.8 asx(CP)

The pre-emerge 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).

Note that the manual page is incorrect in stating that **asx**(CP) supports 386 code. **asx** supports only 8086 and 80286 code.

3.9 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.

3.10 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 usingthe **-Me** flag to **cc**(CP).

3.11 types.h

Some of the C language *.h* files require that *<sys/types.h>* be included first. The following error message generally occurs when a type is used in an #include file but not declared:

old fashioned initialization

Often, the problem is corrected by including *<sys/types.h>* earlier in the program.

3.12 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.

The alternate **malloc** maintains a separate list of free blocks and is 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**.

3.13 brkctl(S) Library

brkcti(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 8086/80286 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.

XENIX 386 does not support 386 processes which have more than one data segment. For compatibility, a special library (*/lib/386/*S*libbrkctl.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=** 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.

3.14 stackuse(CP)

The **stackuse**(CP) utility is not included in this release. The manual page for **stackuse**(CP) should be disregarded.

4. Operating System Specific Software Notes

4.1 8086 Operating System

4.1.1 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.

4.1.2 Utilities Not Supported

The XENIX 8086 Operating System does not support the execution of large model programs. Therefore, only the small model versions of **cc**(CP), **adb**(CP), lint(CP), and **make**(CP) are supported.

sdb(CP) is not supported on the 8086.

The **-r** option of ld(CP) is not supported on the 8086.

4.1.3 adb restrictions

The program debugger adb(CP) works to patch binaries, runs on processes and *core* files, and sets breakpoints in the XENIX 86 Operating System. It does not perform stack backtraces. Subprocess control does not work on medium model programs. Breakpoints cannot be set on medium model programs.

4.1.4 Stack Size Limitations

The default stack for programs run on an 8086 machine is a variable stack, starting at the top of a full 64K byte data segment. A full 64K of physical memory is allocated for combined stack and data. The stack grows down until it reaches the data. While the use of a variable stack can leave unused memory, it is useful for program development. Use the **-F** flag to specify a fixed stack size. We recommend you use the **-F** flag on final versions to increase the performance of your binaries.

4.1.5 Floating-Point Exceptions

For compatibility reasons, floating-point exceptions (like dividing by zero) are masked within the **libc** libraries. Serious problems arise when **printf**(S) attempts to format the result of a "divide-by-zero". On an unmapped machine such as the 8086, anything may happen, from an infinite loop to a system crash.

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 Large Model Utilities

This release includes large model passes of the C compiler. These are invoked using the **-LARGE** flag with **cc**(CP). Only 286 machines can run the large model passes of the compiler. Refer to the **cc**(CP) manual page, and the chapter "cc: A C Compiler" in the XENIX *C User's Guide* for more information on using large model passes.

If you encounter *segmentation violation* errors while using the large model passes of the compiler, try compiling without using code optimization (**-O**). If the errors persist, use the standard compiler passes instead of the large model passes (do not specify **-LARGE** on the **cc**(CP) command line).

In addition to **cc**(CP), large model **adb**(CP), **cflow**(CP), **lint**(CP), **make**(CP), **nm**(CP), and **yacc**(CP) are supplied with this release.

4.2.3 tfind()

The function **tfind**() is missing from the 8086 libraries. If **tfind**() is referenced in a program comiled with the **-M0** or **-M2** option it will be unresolved when the program is linked.

tfind() is present in the 386 libraries and can be used in programs compiled with the **-M3** option. Note that programs compiled with the **-M3** option can only be executed under the XENIX 386 Operating System.

4.2.4 xdata.h

The file */usr/include/xdata.h*, which is referred to in the **execseg**(S) manual page, is missing from the 286 Development System. It contains the following type definitions and declarations:

```
typedef void (far *excode_t)();
typedef char far *exdata_t;
extern excode_t execseg();
```

4.2.5 ld(CP)

The -S option of ld(CP) only accepts values up to 1016. Any argument greater than 1016 returns the error message "Segment limit set too high."

4.2.6 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.

4.2.7 Floating-Point Exceptions

For compatibility reasons, floating-point exceptions (like dividing by zero) are masked within the libc libraries. Serious problems arise when printf(S) attempts to format the result of a "divide-by-zero". On the 80286, this error results in a reasonable memory fault.

4.2.8 sdb

sdb(CP) does not support debugging impure small model programs.

4.2.9 adb(CP)

adb(CP) disassembles the REP instruction prefix as REPNZ and disassembles the REPNZ prefix as REP.

4.3 80386 Operating System

4.3.1 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.

4.3.2 Return Value From main()

The return value from **main()** in a 386 binary is not passed to **exit()** when a program terminates. Programs that do not call **exit()** explicitly, but rely upon returning a value from **main()** will in fact **exit()** with an undefined status.

4.3.3 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.

4.3.4 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 | sort
```

instead of the **-n** option.

4.3.5 masm386

masm386 address and operand prefix generation is not supported in this release.

4.3.6 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, if the **-F** option to the linker was not used, is 4 Kilobytes.

4.4 ANSI Features For 386

The 386 C compiler has a number of differences from the existing 8086/286 compiler. Most of these differences have been caused by a move towards conformance with the proposed ANSI standard for the C programming language. These differences can be disabled by giving the **-Me** option to **cc**(CP). Note that the default option set in */etc/default/cc* is **M3e** and that, for reasons of backward compatibility, these new features are not normally enabled.

4.4.1 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.

4.4.2 Tokens following #else and #endif

The 386 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. The default warning level is 1 so these messages will not

normally be seen.

4.4.3 Substitution of macro formal parameters within quoted strings

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 86/286 C compiler, do allow this. Consider the following macro definition and use:

```
#define CTRL(c)          ('c' & 0x1f)
putchar(CTRL(g));
```

The existing 86/286 compiler expands this to:

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

But according to the ANSI standard it must be:

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

The **-Me** option to the 386 C compiler maintains compatibility with the existing 86/286 compiler, but displays a warning that a non-standard extension has been used. The ANSI standard defines a mechanism for transforming macro formal parameters into quoted strings (the "stringising" operator) and a mechanism for "pasting" strings. Neither of these have yet been implemented in the 386 C compiler. If your code relies on substituting macro parameters within quoted strings then you must compile with the **-Me** option.

4.4.4 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.

4.5 Differences between 286 and 386 code

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.

4.5.1 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.

4.5.2 Size of Pointers

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

4.5.3 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**.

4.5.4 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 in a 386 program have the same alignment as those in 86/286 programs. Note that this directive is only supported by the 386 C compiler.

4.5.5 masm

In addition to supporting the 386 instruction set, the 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*.

5. Documentation Errata

5.1 cc(CP) Manual Page

In the description of the **-O** flag, it should be noted that the "**l**" option is valid only for 80386 machines.

5.2 Intro(S) Manual Page

In the description of error message 12, the documentation states that the message displayed reads:

Not enough space

The actual message displayed is

Not enough core

5.3 Device Drivers

There are two functions to allow systems programmers to translate physical ROM memory addresses to virtual memory addresses for use in the 386 kernel. **mapphys()** and **unmapphys()** give the device driver writer access to ROM. The syntax is:

pp= mapphys(physaddr, len)

Where *physaddr* is a **char *** variable that contains a physical address and *len* is an integer that specifies the number of bytes to map. The function returns a **char *** value which is the kernel virtual address to use to access the desired area.

unmapphys unmaps memory previously mapped with **mapphys**. The parameters to **unmapphys** are as follows:

unmapphys(pp, len)

Where *pp* is the kernel virtual address gotten from a previous **mapphys()** call. *len* is an integer specifying the number of bytes to unmap. *len* should be the same within a **mapphys/unmapphys** sequence.

The function **physio()** fragments all requests for device I/O into 4 Kbyte sections for purposes of reading and writing. Some devices, such as certain tape drives, do not function correctly under this format. This restriction may be eliminated in the future.

Please see the chapter "Writing Device Drivers" in the XENIX *Programmer's Guide*, Volume 2, for more 386 device driver information.

5.3.1 Device Driver Memory Allocation

There is a new way to allocate memory in device drivers. These new routines allows device driver writers to allocate large amounts of physically contiguous physical memory at run time.

The **db_alloc** and **db_free** calls allocate and deallocate contiguous pieces of memory. In addition, the **db_read** and **db_write** calls provide circular queue management for the allocated memory. Note that the device driver writer may choose to substitute custom read and write routines if they wish to manage the allocated memory differently.

Also, note that these routines are not system calls and are useable only from within the kernel. This limits their use to the development of device drivers and other kernel routines. This functionality is identical for both the 80286 and the 80386.

5.3.2 Writing Device Drivers

In the discussion of **dma_alloc** in Chapter 8 of the XENIX *C User's Guide*, it is stated that **dma_alloc** returns a value of 0 if the channel is allocated and a value of 1 if it is not allocated. This statement is incorrect and should read that **dma_alloc** returns a value of 1 if the channel is allocated and 0 if it is not allocated.

Also, note that there are many references to the statement:

```
char far *
```

in the chapter "Writing Device Drivers" in the XENIX *Programmer's Guide*, Volume 2. If you are using a 386, these references should read:

```
char *
```

This is because the 386 is an unsegmented processor.

5.4 swapadd(S) Manual Page

The **swapadd**(S) utility listed in the Contents and Indexes in the *Programmer's Reference* is not supported in this release, and there is no **swapadd**(S) manual page.

5.5 curses(S) Manual Page

On page 4 of the manual page, the line:

mvwgetstr(y, x, str)

should read:

mvwgetstr(win, y, x, str)

And on page 6, there is a reference to the function **restty()** that should read **resetty()**.

5.6 shutdn(S) Manual Page

On page 1 of the **shutdn**(S) manual page, there is a reference to the parameter **ntsblk** in the function call **shutdn**. The function call parameter should read **nsblk**.

Also, the order of the #include files for **shutdn** reads:

#include <filesys.h>
#include <param.h>
#include <types.h>

The order of the #include files should be:

#include <types.h>
#include <param.h>
#include <filesys.h>

5.7 terminfo(S) Manual Page

On page 3 of the **terminfo**(S) manual page, the function calls:

mvgetstr(y, x)
mvwgetstr(win, y, x)

should read:

mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

Also, on page 5 of the **terminfo**(S) manual page, there are references to the **putc** function. Note that this function call is different from the

standard XENIX macro **putc** supplied in **<stdio.h>**. Programmers must supply their own **putc** function for internal use with **terminfo**.

5.8 execseg(S)

The **execseg**(S) manual page should include a note to the effect that **execseg**(S) is a XENIX specific system call. Programs that use **execseg** must be linked with the **-lx** option.