

/ *Technical Standard*

X/Open Curses, Issue 7

The Open Group



© *November 2009, The Open Group*

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

Technical Standard

X/Open Curses, Issue 7

ISBN: 1-931624-83-6

Document Number: C094

Published in the U.K. by The Open Group, November 2009.

This standard has been prepared by The Open Group Base Working Group. Feedback relating to the material contained within this standard may be submitted by using the web site at <http://austingroupbugs.net> with the Project field set to "Xcurses Issue 7".

Contents

| | | | |
|----------------|----------|--|-----------|
| Chapter | 1 | Introduction | 1 |
| | 1.1 | This Document | 1 |
| | 1.1.1 | Relationship to Previous Issues | 1 |
| | 1.1.2 | Features Introduced in Issue 7 | 2 |
| | 1.1.3 | Features Withdrawn in Issue 7 | 2 |
| | 1.1.4 | Features Introduced in Issue 4 | 2 |
| | 1.2 | Conformance | 3 |
| | 1.2.1 | Base Curses Conformance | 3 |
| | 1.2.2 | Enhanced Curses Conformance | 4 |
| | 1.3 | Terminology | 4 |
| | 1.3.1 | Shaded Text | 5 |
| | 1.4 | Format of Entries | 6 |
| | | | |
| Chapter | 2 | General Information | 9 |
| | 2.1 | Use and Implementation of Interfaces | 9 |
| | 2.1.1 | Use and Implementation of Functions | 9 |
| | 2.1.2 | Use and Implementation of Macros | 9 |
| | 2.2 | The Compilation Environment | 10 |
| | 2.2.1 | The X/Open Name Space (ENHANCED CURSES) | 10 |
| | 2.3 | Data Types | 12 |
| | | | |
| Chapter | 3 | Interface Overview | 13 |
| | 3.1 | Components | 13 |
| | 3.1.1 | Relationship to the XSH Specification | 13 |
| | 3.1.2 | Relationship to the XBD Specification | 13 |
| | 3.2 | Screens, Windows, and Terminals | 14 |
| | 3.3 | Characters | 15 |
| | 3.3.1 | Character Storage Size | 15 |
| | 3.3.2 | Multi-Column Characters | 16 |
| | 3.3.3 | Attributes | 16 |
| | 3.3.4 | Rendition | 16 |
| | 3.3.5 | Non-Spacing Characters | 16 |
| | 3.3.6 | Window Properties | 17 |
| | 3.4 | Conceptual Operations | 18 |
| | 3.4.1 | Screen Addressing | 18 |
| | 3.4.2 | Basic Character Operations | 18 |
| | 3.4.3 | Special Characters | 20 |
| | 3.4.4 | Rendition of Characters Placed into a Window | 21 |
| | 3.5 | Input Processing | 22 |
| | 3.5.1 | Keypad Processing | 22 |
| | 3.5.2 | Input Mode | 23 |
| | 3.5.3 | Delay Mode | 24 |
| | 3.5.4 | Echo Processing | 24 |
| | 3.6 | The Set of Curses Functions | 24 |
| | 3.6.1 | Function Name Conventions | 24 |

| | | |
|-------------------|---|------------|
| 3.6.2 | Function Families Provided..... | 25 |
| 3.7 | Interfaces Implemented as Macros..... | 26 |
| 3.8 | Initialized Curses Environment..... | 27 |
| 3.9 | Synchronous and Networked Asynchronous Terminals..... | 27 |
| Chapter 4 | Curses Interfaces..... | 29 |
| Chapter 5 | Headers..... | 305 |
| | < curses.h>..... | 306 |
| | < term.h>..... | 320 |
| | < unctrl.h>..... | 321 |
| Chapter 6 | Utilities..... | 323 |
| | <i>infocmp</i> | 324 |
| | <i>tic</i> | 328 |
| | <i>tput</i> | 330 |
| | <i>untic</i> | 335 |
| Chapter 7 | Terminfo Source Format (ENHANCED CURSES)..... | 337 |
| 7.1 | Source File Syntax..... | 337 |
| 7.1.1 | Minimum Guaranteed Limits..... | 338 |
| 7.1.2 | Formal Grammar..... | 338 |
| 7.1.3 | Defined Capabilities..... | 340 |
| 7.1.4 | Sample Entry..... | 349 |
| 7.1.5 | Types of Capabilities in the Sample Entry..... | 349 |
| Appendix A | Application Usage..... | 353 |
| A.1 | Device Capabilities..... | 353 |
| A.1.1 | Basic Capabilities..... | 353 |
| A.1.2 | Parameterized Strings..... | 354 |
| A.1.3 | Cursor Motions..... | 355 |
| A.1.4 | Area Clears..... | 356 |
| A.1.5 | Insert/Delete Line..... | 356 |
| A.1.6 | Insert/Delete Character..... | 357 |
| A.1.7 | Highlighting, Underlining, and Visible Bells..... | 358 |
| A.1.8 | Keypad..... | 360 |
| A.1.9 | Tabs and Initialization..... | 360 |
| A.1.10 | Delays..... | 361 |
| A.1.11 | Status Lines..... | 361 |
| A.1.12 | Line Graphics..... | 361 |
| A.1.13 | Color Manipulation..... | 363 |
| A.1.14 | Miscellaneous..... | 364 |
| A.1.15 | Special Cases..... | 365 |
| A.1.16 | Similar Terminals..... | 366 |
| A.2 | Printer Capabilities..... | 366 |
| A.2.1 | Rounding Values..... | 366 |
| A.2.2 | Printer Resolution..... | 366 |
| A.2.3 | Specifying Printer Resolution..... | 367 |
| A.2.4 | Capabilities that Cause Movement..... | 369 |
| A.2.5 | Alternate Character Sets..... | 373 |
| A.2.6 | Dot-Matrix Graphics..... | 374 |
| A.2.7 | Effect of Changing Printing Resolution..... | 375 |

Contents

| | | |
|-------|---|------------|
| A.2.8 | Print Quality | 376 |
| A.2.9 | Printing Rate and Buffer Size | 376 |
| A.3 | Selecting a Terminal | 377 |
| A.4 | Application Usage..... | 377 |
| A.4.1 | Conventions for Device Aliases | 377 |
| A.4.2 | Variations of Terminal Definitions..... | 378 |
| | Glossary | 379 |
| | Index..... | 381 |

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow™ will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including TOGAF™ and UNIX® certification.

Further information on The Open Group is available at www.opengroup.org.

The Open Group has over 20 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/certification.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/bookstore.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published at www.opengroup.org/corrigenda.

This Document

This Technical Standard defines the X/Open Curses interface offered to application programs by X/Open Curses-conformant systems. Readers are expected to be experienced C-language programmers and to be familiar with the XBD specification.

This Technical Standard is structured as follows:

- [Chapter 1](#) introduces Curses, gives an overview of enhancements that have been made to this version, and lists specific interfaces that have been withdrawn. This chapter also defines the requirements for conformance to this document and shows the generic format followed by interface definitions in [Chapter 4](#).

- [Chapter 2](#) describes the relationship between Curses and the C language, the compilation environment, and the X/Open System Interface operating system requirements. It also defines the effect of the interface on the name space for identifiers and introduces the major data types that the interfaces use.
- [Chapter 3](#) gives an overview of Curses. It discusses the use of some of the key data types and gives general rules for important common concepts such as characters, renditions, and window properties. It contains general rules for the common Curses operations and operating modes. This information is implicitly referenced by the interface definitions in [Chapter 4](#). The chapter explains the system of naming the Curses functions and presents a table of function families. Finally, the chapter contains notes regarding use of macros and restrictions on block-mode terminals.
- [Chapter 4](#) defines the Curses functional interfaces.
- [Chapter 5](#) defines the contents of headers which declare the functions and global variables, and define types, constants, macros, and data structures that are needed by programs using the services provided by [Chapter 4](#).
- [Chapter 6](#) replaces the specification of the *tput* utility in the XCU specification and defines additional Curses utilities.
- [Chapter 7](#) discusses the **terminfo** database, which Curses uses to describe terminals. The chapter specifies the source format of a **terminfo** entry using a formal grammar, an informal discussion, and an example. Boolean, numeric, and string capabilities are presented in tabular form.
- [Appendix A](#) discusses the use of these capabilities by the writer of a **terminfo** entry to describe the characteristics of the terminal in use.

The chapters are followed by a glossary, which contains normative definitions of terms used in the document. Comprehensive references are available in the index.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures, and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
 - Command operands, command option-arguments, or variable names; for example, substitutable argument prototypes
 - Environment variables, which are also shown in capitals
 - Utility names
 - External variables, such as *errno*
 - Functions; these are shown as follows: *name()*; names without parentheses are C external variables, C function family names, utility names, command operands, or command option-arguments
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.

- Names surrounded by braces—for example, {ARG_MAX}—represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **define** construct.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples, and user input in interactive examples are shown in *fixed width* font. Brackets shown in this font, [], are part of the syntax and do *not* indicate optional items. In syntax the | symbol is used to separate alternatives, and ellipses (. . .) are used to show that additional arguments are optional.
- **Bold fixed width** font is used to identify brackets that surround optional items in syntax, [], and to identify system output in interactive examples.
- Variables within syntax statements are shown in *italic fixed width font*.
- Ranges of values are indicated with parentheses or brackets as follows:
 - (a,b) means the range of all values from a to b , including neither a nor b .
 - $[a,b]$ means the range of all values from a to b , including a and b .
 - $[a,b)$ means the range of all values from a to b , including a , but not b .
 - $(a,b]$ means the range of all values from a to b , including b , but not a .
- Shading is used to identify X/Open Enhanced Curses material, relating to interfaces included to provide enhanced capabilities for applications originally written to be compiled on systems based on the UNIX operating system. Therefore, the features described may not be present on systems that conform to XPG4 or to earlier XPG releases. The relevant reference pages may provide additional or more specific portability warnings about use of the material.

If an entire **SYNOPSIS** section is shaded and marked with EC, all the functionality described in that entry is an extension.

The material on pages labeled ENHANCED CURSES and the material flagged with the EC margin legend is available only in cases where the `_XOPEN_CURSES` version test macro is defined.

Notes:

1. Symbolic limits are used in this document instead of fixed values for portability. The values of most of these constants are defined in `<limits.h>` or `<unistd.h>`.
2. The values of errors are defined in `<errno.h>`.

Trademarks

AT&T[®] is a registered trademark of AT&T in the U.S.A. and other countries.

Boundaryless Information Flow[™] and TOGAF[™] are trademarks and Motif[®], Making Standards Work[®], OSF/1[®], The Open Group[®], UNIX[®], and the “X” device are registered trademarks of The Open Group in the United States and other countries.

Hewlett-Packard[®], HP[®], HP-UX[®], and Openview[®] are registered trademarks of Hewlett-Packard Company.

The names of terminals and of terminal manufacturers cited as examples in [Chapter 7](#) and [Appendix A](#) may be trademarks, which are the property of their respective owners.

Acknowledgements

The Open Group gratefully acknowledges:

- Novell, Inc. for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 4.2 documentation.
- Hewlett-Packard Company, International Business Machines Corporation, Novell Inc., The Open Software Foundation, and Sun Microsystems, Inc., for their work in developing the X/Open UNIX Extension and sponsoring it through the X/Open Direct Review (Fast-track) process.

Referenced Documents

The following documents are referenced in this Technical Standard:

ANSI C

American National Standard for Information Systems: Standard X3.159-1989, Programming Language C.

ISO/IEC 646

ISO/IEC 646: 1991, Information Processing — ISO 7-bit Coded Character Set for Information Interchange.

ISO/IEC 6429: 1992

Information Technology — Control Functions for Coded Character Sets.

ISO/IEC 10646

ISO/IEC 10646-1: 1993, Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.

ISO 2022

ISO 2022: 1986, Information Processing — ISO 7-bit and 8-bit Coded Character Sets — Coded Extension Techniques.

ISO 8859-1

ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

ISO/IEC 9899: 1990

ISO/IEC 9899: 1990, Programming Languages — C, including Amendment 1: 1995 (E), C Integrity (Multibyte Support Extensions (MSE) for ISO C).

SVID, Issue 2

American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue 2; Morristown, NJ, UNIX Press, 1986.

SVID, Issue 3

American Telephone and Telegraph Company, System V Interface Definition (SVID), Issue 3; Morristown, NJ, UNIX Press, 1989.

System V Release 2.0

- UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2).
- UNIX System V Release 2.0 Programming Guide (April 1984 - Issue 2).

System V Release 4.2

Operating System API Reference, UNIX[®] SVR4.2 (1992) (ISBN: 0-13-017658-3).

The following documents published by The Open Group are referenced in this Technical Standard:

Base Specifications, Issue 5

Technical Standard, February 1997, published by The Open Group:

- System Interface Definitions (XBD), Issue 5 (ISBN: 1-85912-186-1, C605)

Referenced Documents

- Commands and Utilities (XCU), Issue 5 (ISBN: 1-85912-191-8, C604)
- System Interfaces and Headers (XSH), Issue 5 (ISBN: 1-85912-181-0, C606)

Base Specifications, Issue 6

Technical Standard, April 2004, published by The Open Group:

- Base Definitions (XBD), Issue 6 (ISBN: 1-931624-43-7, C046)
- System Interfaces (XSH), Issue 6 (ISBN: 1-931624-44-5, C047)
- Shell and Utilities (XCU), Issue 6 (ISBN: 1-931624-45-3, C048)
- Rationale (XRAT), Issue 6 (ISBN: 1-931624-46-1, C049)

Base Specifications, Issue 7

Technical Standard, December 2008, Base Specifications, Issue 7 (ISBN: 1-931624-79-8, C082), published by The Open Group.

Issue 2

X/Open Portability Guide, January 1987, Volume 3: System V Specification Supplementary Definitions: XVS Terminal Interface (ISBN: 0-444-70176-1).

Issue 3

X/Open Specification, February 1992, Supplementary Definitions, Issue 3 (ISBN: 1-872630-38-3, C213), Chapters 9 to 14 inclusive, Curses Interface; this specification was formerly X/Open Portability Guide, Issue 3, Volume 3, January 1989, XSI Supplementary Definitions (ISBN: 0-13-685850-3, XO/XPG/89/004).

Issue 4

CAE Specification, January 1995, X/Open Curses, Issue 4 (ISBN: 1-85912-077-6, C437), published by The Open Group.

Issue 4, Version 2

CAE Specification, July 1996, X/Open Curses, Issue 4, Version 2 (ISBN: 1-85912-171-3, C610), published by The Open Group.

Issue 7

This standard.

The Curses interface provides a terminal-independent method of updating character screens.

The functions in this document are oriented towards locally-connected asynchronous terminals that recognize the codeset of the current locale. For such terminals, applications conforming to this interface are portable. The Curses interface may also be used with synchronous and networked asynchronous terminals, provided the restrictions described in [Section 3.9](#) (on page 27) are considered.

1.1 This Document

This document is Issue 7.

1.1.1 Relationship to Previous Issues

Relationship to Issue 3

The unshaded material in this document preserves syntactic compatibility with the **Curses** specification, Issue 3, except that some functions have been withdrawn (see [Section 1.1.3](#), on page 2). In addition, retained interfaces from the **Curses** specification, Issue 3 have been clarified as a result of industry feedback.

Relationship between Issue 4, Version 1 and Issue 4, Version 2

Version 2 contains corrections and clarifications which have been suggested by industry feedback. In particular, many of the function prototypes have been corrected, and color handling has been further clarified. The CHANGE HISTORY section of the reference pages gives specific detail on when changes were made.

Relationship between Issue 4, Version 2 and Issue 7

Issue 7 is updated as follows:

- Functionality marked “To Be Withdrawn” is removed.
- Clarification is added to explain that the **int** arguments passed to *getbegyx()*, *getmaxyx()*, *getparyx()*, and *getyx()* must be modifiable lvalues.
- The *tparm()* function is marked obsolescent.
- Features described in [Section 1.1.2](#) (on page 2) are introduced.

1.1.2 Features Introduced in Issue 7

The following features are introduced in Issue 7:

- Function prototypes are updated to use **const** where appropriate.
- The *tiparm()* function is added.
- The following new utilities are added in [Chapter 6](#):

infocmp
tic
tput
untic

1.1.3 Features Withdrawn in Issue 7

The following interfaces are withdrawn in this document:

| Withdrawn Interfaces | | | |
|----------------------|------------------|------------------|-------------------|
| <i>tgetent()</i> | <i>tgetnum()</i> | <i>tgoto()</i> | <i>vwprintw()</i> |
| <i>tgetflag()</i> | <i>tgetstr()</i> | <i>vwscanw()</i> | |

1.1.4 Features Introduced in Issue 4

The following features were introduced in Issue 4.

Internationalization

This version of the **Curses** specification has been enhanced to support a wide range of internationalized capabilities. Traditional single-byte character operations are preserved, and multi-byte and wide-character interfaces are included to allow use of the Curses features with a wide range of character codesets. The actual codesets supported are implementation-defined.

Enhanced Character Sets

Emerging character set standards specify characters with a constant width greater than an octet (such as ISO/IEC 10646-1:1993), or multi-byte codesets (such as the ISO 2022:1986 EUC encoding used to encode the Japanese and Chinese language characters).

The previous version of the **Curses** specification was capable of supporting ISO 8859-1:1987. Many traditional implementations only supported ISO/IEC 646:1991 and preceding codeset specifications, in which the length of a character was an octet.

The primary standardization issue with the increasing size of a character is that neither the ANS X3.159-1989 or ISO/IEC 9899:1990 C language definition requires the existence of an integral data type greater than 32-bits. Although such data types are commonly defined, The Open Group cannot require support for them at this time. The opaque data type **cchar_t** and associated routines address this issue.

Writing Direction

The references to writing direction have been generalized to permit both right-to-left and left-to-right writing. This document does not specify whether the implementation supports more than one direction of writing. The behavior of the interfaces in this document is unspecified if the writing direction is vertical, or if the writing direction is horizontal with row height greater than one.

Wide and Non-spacing Characters

New interfaces are introduced for use with wide characters and wide-character strings. The traditional single-byte character string interfaces have been made more general for use with multi-byte character strings. The traditional **chtype** interfaces note that they are usable only in restricted environments and do not support extensible attributes. The behavior of the **chtype** interfaces in this document is unspecified if the **char** data type is greater than 8 bits, or if any single byte character takes more than one display column, or if the application or implementation stores a multi-byte or wide-character value into a **chtype** object.

A new, extensible attribute model has been provided for wide-character interfaces. The display model has been generalized to support both multi-column characters and non-spacing characters. The concept of a complex character is introduced.

Other Enhancements

New interfaces and capabilities are introduced to support color terminals, printers, modems, and mice.

1.2 Conformance

An implementation conforming to this document shall meet the requirements specified by Base Curses conformance (see [Section 1.2.1](#)) or by Enhanced Curses conformance (see [Section 1.2.2](#), on page 4).

1.2.1 Base Curses Conformance

An implementation that claims Base Curses conformance shall meet the following criteria:

- The system shall support all the interfaces and headers defined within this document except that it need not support those occurring on reference pages labeled ENHANCED CURSES and in shaded areas marked with the EC margin legend.
- The **chtype** data type shall support at least octet-based codesets, such as ISO 8859-1: 1987.
- The system may provide additional or enhanced interfaces, headers, and facilities not required by this document, provided that such additions or enhancements do not affect the behavior of an application that requires only the facilities described in this document.

1.2.2 Enhanced Curses Conformance

An implementation that claims Enhanced Curses conformance shall meet the following criteria:

- The system shall support Base Curses conformance as defined above.
- The system shall support the requirements in this document occurring on reference pages labeled ENHANCED CURSES and in shaded areas marked with the EC margin legend.
- The system may provide additional or enhanced interfaces, headers, and facilities not required by this document, provided that such additions or enhancements do not affect the behavior of an application that requires only the facilities described in this document.

1.3 Terminology

The following terms are used in this document:

can

Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

implementation-defined

Describes a value or behavior that is not defined by this document but is selected by an implementor. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

legacy

Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

may

Describes a feature or behavior that is optional for an implementation that conforms to this document. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

must

Describes a feature or behavior that is mandatory for an application or user. An implementation that conforms to this document shall support this feature or behavior.

shall

Describes a feature or behavior that is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

should

For an implementation that conforms to this document, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the

feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

undefined

Describes the nature of a value or behavior not defined by this document which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by this document which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

will

Same meaning as *shall*; *shall* is the preferred term.

1.3.1 Shaded Text

Shaded text in this document is qualified by a code in the left margin. The codes and their meanings are as follows:

EC

X/Open Enhanced Curses

The functionality described relates to interfaces included to provide enhanced capabilities for applications originally written to be compiled on systems based on the UNIX operating system. Therefore, the features described may not be present on systems that conform to XPG4 or to earlier XPG releases. The relevant reference pages may provide additional or more specific portability warnings about use of the material.

If an entire SYNOPSIS section is shaded and marked EC, all the functionality described on that reference page is an extension.

The functionality on reference pages labeled ENHANCED CURSES and the functionality flagged with the EC margin legend are available only in cases where the `_XOPEN_CURSES` version test macro is defined.

OB

Obsolescent

The functionality described may be removed in a future version of this document. Applications should not use obsolescent features.

Where applicable, the material is identified by use of the OB margin legend.

1.4 Format of Entries

The entries in [Chapter 4](#) and [Chapter 5](#) are based on a common format, as follows. The only sections relating to conformance are the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS sections.

NAME

This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS

This section summarizes the use of the entry being described. If it is necessary to include a header to use this interface, the names of such headers are shown; for example:

```
#include <stdio.h>
```

DESCRIPTION

This section describes the functionality of the interface or header.

RETURN VALUE

This section indicates the possible return values, if any.

If the implementation can detect errors, “successful completion” means that no error has been detected during execution of the function. If the implementation does detect an error, the error is indicated.

For functions where no errors are defined, “successful completion” means that if the implementation checks for errors, no error has been detected. If the implementation can detect errors, and an error is detected, the indicated return value will be returned and *errno* may be set.

ERRORS

This section gives the symbolic names of the error values returned by a function or stored into a variable accessed through the symbol *errno* if an error occurs.

“No errors are defined” means that error values returned by a function or stored into a variable accessed through the symbol *errno*, if any, depend on the implementation.

EXAMPLES

This section is informative.

This section gives examples of usage, where appropriate.

APPLICATION USAGE

This section is informative.

This section gives warnings and advice to application developers about the entry.

RATIONALE

This section is informative.

This section contains historical information concerning the contents of the entry and why features were included or discarded by the developers of this document.

FUTURE DIRECTIONS

This section is informative.

This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO

This section is informative.

This section gives references to related information.

CHANGE HISTORY

This section is informative.

This section shows the derivation of the entry and any significant changes that have been made to it.

The entries in [Chapter 6](#) are in the same format as the utility reference pages in the XCU specification (see the XCU specification, Section 1.4, Utility Description Defaults).

2.1 Use and Implementation of Interfaces

2.1.1 Use and Implementation of Functions

Each of the following statements shall apply to all functions unless explicitly stated otherwise in the detailed descriptions that follow:

1. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.
2. Any function declared in a header may also be implemented as a macro defined in the header, so a function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a function even if it is also defined as a macro. The use of the C-language `#undef` construct to remove any such macro definition shall also ensure that an actual function is referred to.
3. Any invocation of a function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.
4. Provided that a function can be declared without reference to any type defined in a header, it is also permissible to declare the function explicitly and use it without including its associated header.
5. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

2.1.2 Use and Implementation of Macros

Each of the following statements shall apply to all macros unless explicitly stated otherwise:

1. Any definition of an object-like macro in a header shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.
2. Any definition of a function-like macro in a header shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so that it is generally safe to use arbitrary expressions as arguments.

3. Any definition of a function-like macro in a header can be invoked in an expression anywhere a function with a compatible return type could be called.

2.2 The Compilation Environment

The compilation environment in this document can exist in the following environment:

- Base Specifications, Issue 7

The compilation environment is defined as follows:

Applications shall ensure that the feature test macro `_XOPEN_SOURCE` is defined with the value 700 before inclusion of any header. This is needed to enable the functionality described in this document, and possibly to enable functionality defined elsewhere in the Common Applications Environment.

In the compilation of an application that **#defines** the `_XOPEN_SOURCE` feature test macro, no header defined by this document or by the Base Specifications, Issue 7 shall be included prior to the definition of the feature test macro. This restriction also applies to any implementation-defined header in which these feature test macros are used. If the definition of the macro does not precede the **#include**, the result is undefined.

Identifiers in this document may only be undefined using the **#undef** directive as described in [Section 2.1.1](#) (on page 9) or [Section 2.2.1](#). These **#undef** directives must follow all **#include** directives of any XSI headers.

Since this document is aligned with the ISO C Standard, and since all functionality enabled by `_POSIX_C_SOURCE` set equal to 200809L is enabled by `_XOPEN_SOURCE` set equal to 700, there should be no need to define `_POSIX_C_SOURCE` if `_XOPEN_SOURCE` is so defined. Therefore, if `_XOPEN_SOURCE` is set equal to 700 and `_POSIX_C_SOURCE` is set equal to 200809L, the behavior is the same as if only `_XOPEN_SOURCE` is defined and set equal to 700. However, should `_POSIX_C_SOURCE` be set to a value greater than 200809L, the behavior is unspecified.

The `c99` utility shall recognize the following additional `-I` option for standard libraries:

-I curses This option shall make available all interfaces referenced in this document (except for those labeled ENHANCED CURSES and except for portions marked with the EC margin legend).

EC If the implementation defines `_XOPEN_CURSES`, then **-I curses** shall also make available all interfaces referenced in this document and labeled ENHANCED CURSES and portions marked with the EC margin legend.

It is unspecified whether the library `libcurses.a` exists as a regular file.

2.2.1 The X/Open Name Space (ENHANCED CURSES)

EC The requirements in this section are in effect only for implementations that claim Enhanced Curses compliance.

All identifiers in this document are defined in at least one of the headers, as shown in [Chapter 5](#). When `_XOPEN_SOURCE` is defined, each header defines or declares some identifiers, potentially conflicting with identifiers used by the application. The set of identifiers visible to the application consists of precisely those identifiers from the header pages of the included headers, as well as additional identifiers reserved for the implementation. In addition, some headers may

make visible identifiers from other headers as indicated on the relevant header pages.

Implementations may also add members to a structure or union without controlling the visibility of those members with a feature test macro, as long as a user-defined macro with the same name cannot interfere with the correct interpretation of the program.

The identifiers reserved for use by the implementation are described below:

1. Each identifier with external linkage described in the header section is reserved for use as an identifier with external linkage if the header is included.
2. Each macro name described in the header section is reserved for any use if the header is included.
3. Each identifier with file scope described in the header section is reserved for use as an identifier with file scope in the same name space if the header is included.
4. All identifiers consisting of exactly two (2) uppercase letters.

If any header is included, identifiers with the `_t` suffix are reserved for any use by the implementation.

If any header in the following table is included, macros with the prefixes shown may be defined. After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes for its own purpose, provided their use is preceded by an `#undef` of the corresponding macro.

| Header | Prefix |
|--------------------------------|---|
| <code><courses.h></code> | <code>A_, ACS_, ALL_, BUTTON, COLOR_, KEY_, MOUSE, REPORT_, WA_, WACS_</code> |
| <code><term.h></code> | <code>ext_</code> |

The following identifiers are reserved regardless of the inclusion of headers:

1. With the exception of identifiers beginning with the prefix `_POSIX_`, all identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use by the implementation.
2. All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.
3. All identifiers listed as reserved in the **XSH** specification are reserved for use as identifiers with external linkage.
4. All the identifiers defined in this document that have external linkage are always reserved for use as identifiers with external linkage.

No other identifiers are reserved.

Applications must not declare or define identifiers with the same name as an identifier reserved in the same context. Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names must not be defined if any associated header is included.

Headers may be included in any order, and each may be included more than once in a given scope, with no difference in effect from that of being included only once.

If used, a header must be included outside of any external declaration or definition, and it must be first included before the first reference to any type or macro it defines, or to any function or object it declares. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. Prior to the inclusion of a header, the program must not define any macros with

names lexically identical to symbols defined by that header.

2.3 Data Types

All of the data types used by Curses functions are defined by the implementation. The following list describes these types:

| | | |
|----|----------------|--|
| EC | attr_t | An integer type that can contain at least an unsigned short . The type attr_t is used to hold an OR'ed set of attributes defined in <code><curses.h></code> that begin with the prefix <code>WA_</code> . |
| | bool | As described in <code><stdbool.h></code> . |
| | chtype | An integer type that can contain at least an unsigned char and attributes. Values of type chtype are formed by OR'ing together an unsigned char value and zero or more of the base attribute flags defined in <code><curses.h></code> that have the <code>A_</code> prefix. The application can extract these components of a chtype value using the base masks defined in <code><curses.h></code> for this purpose. |
| EC | | The chtype data type also contains a color-pair. Values of type chtype are formed by OR'ing together an unsigned char value, a color pair, and zero or more of the attributes defined in <code><curses.h></code> that begin with the prefix <code>A_</code> . The application can extract these components of a chtype value using the masks defined in <code><curses.h></code> for this purpose. |
| | SCREEN | An opaque terminal representation. |
| EC | wint_t | As described in <code><wchar.h></code> . |
| EC | wchar_t | As described in <code><stddef.h></code> . |
| EC | cchar_t | A type that can reference a string of wide characters of up to an implementation-defined length, a color-pair, and zero or more attributes from the set of all attributes defined in this document. A null cchar_t object is an object that references an empty wide-character string. Arrays of cchar_t objects are terminated by a null cchar_t object. |
| | WINDOW | An opaque window representation. |

3.1 Components

A Curses initialization function, usually *initscr()*, determines the terminal model in use, by reference to either an argument or an environment variable. If that model is defined in **terminfo**, then the same **terminfo** entry tells Curses exactly how to operate the terminal.

In this case, a comprehensive API lets the application perform terminal operations. The Curses runtime system receives each terminal request and sends appropriate commands to the terminal to achieve the desired effect.

3.1.1 Relationship to the XSH Specification

Error Numbers

Most functions provide an error number in *errno*, which is a symbol defined or declared in **<errno.h>** as either a macro or an identifier declared with external linkage; the symbol expands to a modifiable lvalue of type **int**.

A list of valid values for *errno* and advice to application writers on the use of *errno* appears in the **XSH** specification.

Signals

Curses implementations may provide for special handling of the SIGINT, SIGQUIT, and SIGTSTP signals if their disposition is SIG_DFL at the time *initscr()* is called (see *initscr()*, on page 126).

Any special handling for these signals may remain in effect for the life of the process or until the process changes the disposition of the signal.

None of the Curses functions are required to be safe with respect to signals (see *sigaction()* in the **XSH** specification).

Thread-Safety

The interfaces defined by this document need not be thread-safe.

3.1.2 Relationship to the XBD Specification

Applications using Curses should not also control the terminal using capabilities of the general terminal interface defined in the **XBD** specification, Chapter 11, General Terminal Interface.

There is no requirement that the paradigms that exist while in Curses mode be carried over outside the Curses environment (see *def_prog_mode()*).

Signals

The behavior of Curses with respect to signals not defined by the XBD specification is unspecified.

3.2 Screens, Windows, and Terminals

Screen

A screen is the physical output device of the terminal. In Curses, a **SCREEN** data type is an opaque data type associated with a terminal. Each window (see below) is associated with a **SCREEN**.

Window

The Curses functions permit manipulation of *window* objects, which can be thought of as two-dimensional arrays of characters and their renditions representing all or part of a terminal's physical screen. Windows do not have to correspond to the entire screen. It is possible to create smaller windows and also to indicate that a window is only partially visible on the screen. It is possible to create windows larger than the terminal screen using pads. A default window called *stdscr*, which is the size of the terminal screen, is supplied. Others may be created with *newterm()*.

Data structures declared as **WINDOW** refer to windows (and to subwindows, derived windows, pads, and subpads, as described elsewhere). These data structures are manipulated with functions described in [Chapter 7](#).

Among the most basic functions are *move()* and *addch()* which manipulate the default window *stdscr*, and *refresh()* which tells Curses to update the user's screen from *stdscr*. More general versions of these functions enable specific windows to be manipulated and refreshed.

Line drawing characters may be specified to be output. On input, Curses is also able to translate arrow and function keys that transmit escape sequences into single values. The line drawing characters and input values use names defined in `<curses.h>`.

Each window has a flag that indicates that the information in the window could differ from the information displayed on the terminal device. Making any change to the contents of the window, moving or modifying the window, or setting the window's cursor position, sets this flag (*touches* the window). Refreshing the window clears this flag. (For further information, see [is_linetouched\(\)](#) (on page 141).)

Subwindow

A *subwindow* is a window, created within another window (called the *parent window*), and positioned relative to the parent window. A subwindow can be created by calling *derwin()*, *newpad()*, or *subwin()*.

Subwindows can be created from a parent window by calling *subwin()*. The position and size of subwindows on the screen must be identical to or totally within the parent window. Changes to either the parent window or the subwindow affect both. Window clipping is not a property of subwindows.

Ancestor

The term *ancestor* refers to a window's parent, or its parent, and so on.

Derived Window

Derived windows are subwindows whose position is defined by reference to the parent window rather than in absolute screen coordinates. Derived windows are otherwise no different from subwindows.

Pad

A pad is a specialized case of a window which can be bigger than the actual screen size and is not necessarily associated with a particular part of the screen. Pads should be used whenever a window larger than the terminal screen is required.

Subpad

A subpad is a specialized case of a window created within another pad.

Terminal

A terminal is the logical input and output device through which character-based applications interact with the user. **TERMINAL** is an opaque data type associated with a terminal. A **TERMINAL** data structure primarily contains information about the capabilities of the terminal, as defined by **terminfo**. A **TERMINAL** also contains information about the terminal modes and current state for input and output operations. Each screen (see above) is associated with a **TERMINAL**.

3.3 Characters

3.3.1 Character Storage Size

Historically, a position on the screen has corresponded to a single stored byte. This correspondence is no longer true for several reasons:

- Some characters may occupy several columns when displayed on the screen (see [Section 3.3.2](#), on page 16).
- Some characters may be non-spacing characters, defined only in association with a spacing character (see [Section 3.3.5](#), on page 16).
- The number of bytes to hold a character from the extended character sets depends on the `LC_CTYPE` locale category.

The internal storage format of characters and renditions is unspecified. There is no implied correspondence between the internal storage format and the external representation of characters and renditions in objects of type **chtype** and **cchar_t**.

3.3.2 Multi-Column Characters

EC Some character sets define *multi-column characters* that occupy more than one column position when displayed on the screen.

Writing a character whose width is greater than the width of the destination window is an error.

3.3.3 Attributes

Each character can be displayed with *attributes* such as underlining, reverse video, or color on terminals that support such display enhancements. Current attributes of a window are applied to all characters that are written into the window with *waddch()*, *wadd_wch()*, *waddstr()*, *waddchstr()*, *waddwstr()*, *waddwchstr()*, and *wprintw()*. Attributes can be combined.

EC Attributes can be specified using constants with the A_ prefix specified in **<curses.h>**. The A_ constants manipulate attributes in objects of type **chtype**. Additional attributes can be specified using constants with the WA_ prefix. The WA_ constants manipulate attributes in objects of type **attr_t**.

Two constants that begin with A_ and WA_ and that represent the same terminal capability refer to the same attribute in the **terminfo** database and in the window data structure. The effect on a window does not differ depending on whether the application specifies A_ or WA_ constants. For example, when an application updates window attributes using the interfaces that support the A_ values, a query of the window attribute using the function that returns WA_ values reflects this update. When it updates window attributes using the interfaces that support the WA_ values, for which corresponding A_ values exist, a query of the window attribute using the function that returns A_ values reflects this update.

3.3.4 Rendition

EC The *rendition* of a character displayed on the screen is its attributes and a color pair.

The rendition of a character written to the screen becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations. To the extent possible on a particular terminal, a character's rendition corresponds to the graphic rendition of the character put on the screen.

If a given terminal does not support a rendition that an application program is trying to use, Curses may substitute a different rendition for it.

EC Colors are always used in pairs (referred to as *color-pairs*). A color-pair consists of a foreground color (for characters) and a background color (for the field on which the characters are displayed).

3.3.5 Non-Spacing Characters

EC The requirements in this section are in effect only for implementations that claim Enhanced Curses compliance.

Some character sets may contain *non-spacing* characters. (Non-spacing characters are those, other than the '\0' character, for which *wcwidth()* returns a width of zero.) The application may write non-spacing characters to a window. Every non-spacing character in a window is associated with a spacing character and modifies the spacing character. Non-spacing characters

in a window cannot be addressed separately. A non-spacing character is implicitly addressed whenever a Curses operation affects the spacing character with which the non-spacing character is associated.

Non-spacing characters do not support attributes. For interfaces that use wide characters and attributes, the attributes are ignored if the wide character is a non-spacing character. Multi-column characters have a single set of attributes for all columns. The association of non-spacing characters with spacing characters can be controlled by the application using the wide-character interfaces. The wide-character string functions provide codeset-dependent association.

Two typical effects of a non-spacing character associated with a spacing character called 'c' are as follows:

- The non-spacing character may modify the appearance of 'c'. (For instance, there may be non-spacing characters that add diacritical marks to characters. However, there may also be spacing characters with built-in diacritical marks.)
- The non-spacing character may bridge 'c' to the character following 'c'. (Examples of this usage are the formation of ligatures and the conversion of characters into compound display forms, words, or ideograms.)

Implementations may limit the number of non-spacing characters that can be associated with a spacing character, provided any limit is at least five (5).

Complex Characters

A *complex character* is a set of associated characters, which may include a spacing character and may include any non-spacing characters associated with it. A *spacing complex character* is a spacing character followed by any non-spacing characters associated with it; that is, a spacing complex character is a complex character that includes one spacing character. An example of a code set that has complex characters is ISO/IEC 10646-1:1993.

A complex character can be written to the screen; if it does not include a spacing character, any non-spacing characters are associated with the spacing complex character that exists at the specified screen position. When the application reads information back from the screen, it obtains spacing complex characters.

The `cchar_t` data type represents a complex character and its rendition. When a `cchar_t` represents a non-spacing complex character (that is, when there is no spacing character within the complex character), then its rendition is not used; when it is written to the screen, it uses the rendition specified by the spacing character already displayed.

An object of type `cchar_t` can be initialized using `setcchar()` and its contents can be extracted using `getcchar()`. The behavior of functions that take a `cchar_t` input argument is undefined if the application provides a `cchar_t` value that was not initialized in this way or obtained from a Curses function that has a `cchar_t` output argument.

3.3.6 Window Properties

Associated with each window are the following properties that affect the placing of characters into the window (see [Section 3.4.4](#), on page 21).

Window Rendition

Each window has a rendition, which is combined with the rendition component of the window's background property described below.

Window Background

Each window has a background property. The background property specifies:

- A spacing complex character (the background character) that will be used in a variety of situations where visible information is deleted from the screen
- A rendition to use in displaying the background character in those situations, and in other situations specified in [Section 3.4.4](#) (on page 21)

3.4 Conceptual Operations

3.4.1 Screen Addressing

Many Curses functions use a coordinate pair. In the DESCRIPTION, coordinate locations are represented as (y, x) since the y argument always precedes the x argument in the function call. These coordinates denote a line/column position, not a character position.

The coordinate y always refers to the row (of the window), and x always refers to the column. The first row and the first column is number 0, not 1. The position $(0, 0)$ is the window's *origin*.

For example, for terminals that display the ISO 8859-1:1987 character set (with left-to-right writing), $(0, 0)$ represents the upper left-hand corner of the screen.

Functions that start with *mv* take arguments that specify a (y, x) position and move the cursor (as though *move()* were called) before performing the requested action. As part of the requested action, further cursor movement may occur, specified on the respective reference page.

3.4.2 Basic Character Operations

Adding (Overwriting)

EC The Curses functions that contain the word *add*—such as *addch()*—actually specify one or more characters to replace (overwrite) characters already in the window. If these functions specify only non-spacing characters, they are appended to a spacing character already in the window; see also [Section 3.3.5](#) (on page 16).

When replacing a multi-column character with a character that requires fewer columns, the new character is added starting at the specified or implied column position. All columns that the former multi-column character occupied that the new character does not require are *orphaned columns*, which are filled using the background character and rendition.

Replacing a character with a character that requires more columns also replaces one or more subsequent characters on the line. This process may also produce orphaned columns.

Truncation, Wrapping, and Scrolling

EC If the application specifies a character or a string of characters such that writing them to a window would extend beyond the end of the line (for example, if the application tries to deposit any multi-column character at the last column in a line), the behavior depends on whether the function supports line wrapping:

- If the function does not wrap, it fails.
- If the function wraps, then it places one or more characters in the window at the start of the next line, beginning with the first character that would not completely fit on the original line.

EC If the final character on the line is a multi-column character that does not completely fit on the line, the entire character wraps to the next line and columns at the end of the original line may be orphaned.

If the original line was the last line in the window, the wrap may cause a scroll to occur:

- If scrolling is enabled, a scroll occurs. The contents of the first line of the window are lost. The contents of each remaining line in the window move to the previous line. The last line of the window is filled with any characters that wrapped. Any remaining space on the last line is filled with the background character and rendition.
- If scrolling is disabled, any characters that would extend beyond the last column of the last line are truncated.

The *scrollok()* function enables and disables scrolling.

Some *add* functions move the cursor just beyond the end of the last character added. If this position is beyond the end of a line, it causes wrapping and scrolling under the conditions specified in the second bullet above.

Insertion

Insertion functions (such as *insch()*) insert characters immediately before the character at the specified or implied cursor position.

The insertion shifts all characters that were formerly at or beyond the cursor position on the cursor line toward the end of that line. Since none of the insertion functions support wrapping, the characters that would thus extend beyond the end of the line are removed from the window. This may produce orphaned columns.

EC If multi-column characters are displayed, some cursor positions are within a multi-column character but not at the beginning of a character. Any request to insert data at a position that is not the beginning of a multi-column character will be adjusted so that the actual cursor position is at the beginning of the multi-column character in which the requested position occurs.

There are no warning indications relative to cursor relocation. The application should not maintain an image of the cursor position, since this constitutes placing terminal-specific information in the application and defeats the purpose of using Curses.

Portable applications cannot assume that a cursor position specified in an insert function is a reusable indication of the actual cursor position. Portable applications should use *getyx()* to obtain the current cursor position in a window.

Deletion

EC Deletion functions (such as *delch()*) delete the simple or complex character at the specified or implied cursor position, no matter which column of the character this is. All column positions are replaced by the background character and rendition and the cursor is not relocated. If a character-deletion operation would cause a previous wrapping operation to be undone, then the results are unspecified.

Window Operations

EC Overlapping a window (that is, placing one window on top of another) and overwriting a window (that is, copying the contents of one window into another) follows the operation of overwriting multi-column glyphs around its edge. Any orphaned columns are handled as in the character operations.

Characters that Straddle the Subwindow Border

EC A subwindow can be defined such that multi-column characters straddle the subwindow border. The character operations deal with these straddling characters as follows:

- Reading the subwindow with a function such as *in_wch()* reads the entire straddling character.
- Adding, inserting, or deleting in the subwindow deletes the entire straddling character before the requested operation begins and does not relocate the cursor.
- Scrolling lines in the subwindow has the following effects:
 - A straddling character at the start of the line is completely erased before the scroll operation begins.
 - A straddling character at the end of the line moves in the direction of the scroll and continues to straddle the subwindow border. Column positions outside the subwindow at the straddling character's former position are orphaned unless another straddling character scrolls into those positions.

If the application calls a function such as *border()*, the above situations do not occur because writing the border on the subwindow deletes any straddling characters.

In the above cases involving multi-column characters, operations confined to a subwindow can modify the screen outside the subwindow. Therefore, saving a subwindow, performing operations within the subwindow, and then restoring the subwindow may disturb the appearance of the screen. To overcome these effects (for example, for pop-up windows), the application should refresh the entire screen.

3.4.3 Special Characters

Some functions process special characters as specified below.

In functions that do not move the cursor based on the information placed in the window, these special characters would only be used within a string in order to affect the placement of subsequent characters; the cursor movement specified below does not persist in the visible cursor beyond the end of the operation. In functions that do move the cursor, these special characters can be used to affect the placement of subsequent characters and to achieve movement of the visible cursor.

- <backspace> Unless the cursor was already in column 0, <backspace> moves the cursor one column toward the start of the current line and any characters after the <backspace> are added or inserted starting there.
- <carriage-return> Unless the cursor was already in column 0, <carriage-return> moves the cursor to the start of the current line. Any characters after the <carriage-return> are added or inserted starting there.
- <newline> In an add operation, Curses adds the background character into successive columns until reaching the end of the line. Scrolling occurs as described in [Truncation, Wrapping, and Scrolling](#) (on page 19). Any characters after the <newline> character are added, starting at the start of the new line.
- In an insert operation, <newline> erases the remainder of the current line with the background character (effectively a `wclrtoeol()`) and moves the cursor to the start of a new line. When scrolling is enabled, advancing the cursor to a new line may cause scrolling as described in [Truncation, Wrapping, and Scrolling](#) (on page 19). Any characters after the <newline> character are inserted at the start of the new line.
- If **lines** equals one, the behavior is unspecified (note that the `filter()` function sets **lines** equal to one).
- <tab> Tab characters in text move subsequent characters to the next horizontal tab stop. Curses may assume that tab stops are in column 0, 8, 16, and so on.
- In an insert or add operation, Curses inserts or adds, respectively, the background character into successive columns until reaching the next tab stop. If there are no more tab stops in the current line, wrapping and scrolling occur as described in [Truncation, Wrapping, and Scrolling](#) (on page 19).

Control Characters

The Curses functions that perform special-character processing conceptually convert control characters to the caret ('^') character followed by a second character (which is an uppercase letter if it is alphabetic) and write this string to the window in place of the control character. The functions that retrieve text from the window will not retrieve the original control character.

3.4.4 Rendition of Characters Placed into a Window

When the application adds or inserts characters into a window, the effect is as follows:

If the character is not the <space> character, then the window receives:

- The character that the application specifies
- The color that the application specifies; or the window color, if the application does not specify a color
- The attributes specified, OR'ed with the window attributes

If the character is the <space> character, then the window receives:

- The background character
- The color that the application specifies; or the window color, if the application does not specify a color

- The attributes specified, OR'ed with the window attributes

3.5 Input Processing

The Curses input model provides a variety of ways to obtain input from the keyboard.

3.5.1 Keypad Processing

The application can enable or disable *keypad translation* by calling `keypad()`. When translation is enabled, Curses attempts to translate a sequence of terminal input that represents the pressing of a function key into a single key code. When translation is disabled, Curses passes terminal input to the application without such translation, and any interpretation of the input as representing the pressing of a keypad key must be done by the application.

EC The complete set of key codes for keypad keys that Curses can process is specified by the constants defined in `< curses.h >` whose names begin with `KEY_`. Each terminal type described in the `terminfo` database may support some or all of these key codes. The `terminfo` database specifies the sequence of input characters from the terminal type that correspond to each key code (see [Section A.1.8](#), on page 360).

The Curses implementation cannot translate keypad keys on terminals where pressing the keys does not transmit a unique sequence.

When translation is enabled and a character that could be the beginning of a function key (such as escape) is received, Curses notes the time and begins accumulating characters. If Curses receives additional characters that represent the pressing of a keypad key, within an unspecified interval from the time the first character was received, then Curses converts this input to a key code for presentation to the application. If such characters are not received during this interval, translation of this input does not occur and the individual characters are presented to the application separately. (Because Curses waits for this interval to accumulate a key code, many terminals experience a delay between the time a user presses the escape key and the time the escape is returned to the application.)

EC In addition, No Timeout Mode provides that in any case where Curses has received part of a function key sequence, it waits indefinitely for the complete key sequence. The “unspecified interval” in the previous paragraph becomes infinite in No Timeout Mode. No Timeout Mode allows the use of function keys over slow communication lines. No Timeout Mode lets the user type the individual characters of a function key sequence, but also delays application response when the user types a character (not a function key) that begins a function key sequence. For this reason, in No Timeout Mode many terminals will appear to hang between the time a user presses the escape key and the time another key is pressed. No Timeout Mode is switchable by calling `notimeout()`.

If any special characters (see [Section 3.4.3](#), on page 20) are defined or redefined to be characters that are members of a function key sequence, then Curses will be unable to recognize and translate those function keys.

Several of the modes discussed below are described in terms of availability of input. If keypad translation is enabled, then input is not available once Curses has begun receiving a keypad sequence until the sequence is completely received or the interval has elapsed.

3.5.2 Input Mode

The **XBD** specification (Special Characters) defines flow-control characters, the interrupt character, the erase character, and the kill character. Four mutually-exclusive Curses modes let the application control the effect of these input characters:

EC

| Input Mode | Effect |
|--------------------|---|
| Cooked Mode | This achieves normal line-at-a-time processing with all special characters handled outside the application. This achieves the same effect as canonical-mode input processing as specified in the XBD specification. The state of the ISIG and IXON flags are not changed upon entering this mode by calling <code>nocbreak()</code> , and are set upon entering this mode by calling <code>noraw()</code> . The implementation supports erase and kill characters from any supported locale, no matter what the width of the character. |
| <i>cbreak</i> Mode | Characters typed by the user are immediately available to the application and Curses does not perform special processing on either the erase character or the kill character. An application can select <i>cbreak</i> mode to do its own line editing but to let the abort character be used to abort the task. This mode achieves the same effect as non-canonical-mode, Case B input processing (with MIN set to 1 and ICRNL cleared) as specified in the XBD specification. The state of the ISIG and IXON flags are not changed upon entering this mode. |
| Half-Delay Mode | The effect is the same as <i>cbreak</i> , except that input functions wait until a character is available or an interval defined by the application elapses, whichever comes first. This mode achieves the same effect as non-canonical-mode, Case C input processing (with TIME set to the value specified by the application) as specified in the XBD specification. The state of the ISIG and IXON flags are not changed upon entering this mode. |
| Raw Mode | Raw mode gives the application maximum control over terminal input. The application sees each character as it is typed. This achieves the same effect as non-canonical mode, Case D input processing as specified in the XBD specification. The ISIG and IXON flags are cleared upon entering this mode. |

EC

The terminal interface settings are recorded when the process calls `initscr()` or `newterm()` to initialize Curses and restores these settings when `endwin()` is called. The initial input mode for Curses operations is unspecified unless the implementation supports Enhanced Curses compliance, in which case the initial input mode is *cbreak* mode.

The behavior of the BREAK key depends on other bits in the display driver that are not set by Curses.

3.5.3 Delay Mode

Two mutually-exclusive delay modes specify how quickly certain Curses functions return to the application when there is no terminal input waiting when the function is called:

No Delay The function fails.

Delay The application waits until the implementation passes text through to the application. If *cbreak* or Raw Mode is set, this is after one character. Otherwise, this is after the first <newline> character, end-of-line character, or end-of-file character.

The effect of No Delay Mode on function key processing is unspecified.

3.5.4 Echo Processing

Echo mode determines whether Curses echoes typed characters to the screen. The effect of Echo mode is analogous to the effect of the ECHO flag in the local mode field of the **termios** structure associated with the terminal device connected to the window. However, Curses always clears the ECHO flag when invoked, to inhibit the operating system from performing echoing. The method of echoing characters is not identical to the operating system's method of echoing characters, because Curses performs additional processing of terminal input.

If in Echo mode, Curses performs its own echoing: Any visible input character is stored in the current or specified window by the input function that the application called, at that window's cursor position, as though *addch()* were called, with all consequent effects such as cursor movement and wrapping.

If not in Echo mode, any echoing of input must be performed by the application. Applications often perform their own echoing in a controlled area of the screen, or do not echo at all, so they disable Echo mode.

It may not be possible to turn off echo processing for synchronous and networked asynchronous terminals because echo processing is done directly by the terminals. Applications running on such terminals should be aware that any characters typed will appear on the screen at wherever the cursor is positioned.

3.6 The Set of Curses Functions

The Curses functions allow: overall screen, window, and pad manipulation; output to windows and pads; reading terminal input; control over terminal and Curses input and output options; environment query functions; color manipulation; use of soft label keys; access to the **terminfo** database of terminal capabilities; and access to low-level functions.

3.6.1 Function Name Conventions

The reference pages in [Chapter 4](#) present families of multiple Curses functions. Most function families have different functions that give the programmer the following options:

- A function with the basic name operates on the window *stdscr*. A function with the same name plus the *w* prefix operates on a window specified by the *win* argument.

When the reference page for a function family refers to the *current or specified window*, it means *stdscr* for the basic functions and the window specified by *win* for any *w* function.

Functions whose names have the *p* prefix require an argument that is a pad instead of a window.

- A function with the basic name operates based on the current cursor position (of the current or specified window, as described above). A function with the same name plus the *mv* prefix moves the cursor to a position specified by the *y* and *x* arguments before performing the specified operation.

When the reference page for a function family refers to the *current or specified position*, it means the cursor position for the basic functions and the position (*y*, *x*) for any *mv* function.

The *mvw* prefix exists and combines the *mv* semantics discussed here with the *w* semantics discussed above. The window argument is always specified before the coordinates.

- A function with the basic name is often provided for historical compatibility and operates only on single-byte characters. A function with the same name plus the *w* infix operates on wide (multi-byte) characters. A function with the same name plus the *_w* infix operates on complex characters and their renditions.
- When a function with the basic name operates on a single character, there is sometimes a function with the same name plus the *n* infix that operates on multiple characters. An *n* argument specifies the number of characters to process. The respective reference page specifies the outcome if the value of *n* is inappropriate.

3.6.2 Function Families Provided

| Function Names | Description | s | w | c | Refer to |
|-------------------------------------|--|---|---|---|----------------------------|
| | Add (Overwrite) | | | | |
| <code>[mv][w]addch()</code> | Add a character | Y | Y | Y | <code>addch()</code> |
| <code>[mv][w]addch[n]str()</code> | Add a character string | N | N | N | <code>addchstr()</code> |
| <code>[mv][w]add[n]str()</code> | Add a string | Y | Y | Y | <code>addnstr()</code> |
| <code>[mv][w]add[n]wstr()</code> | Add a wide-character string | Y | Y | Y | <code>addnwstr()</code> |
| <code>[mv][w]add_wch()</code> | Add a wide character and rendition | Y | Y | Y | <code>add_wch()</code> |
| <code>[mv][w]add_wch[n]str()</code> | Add an array of wide characters and renditions | ? | N | N | <code>add_wchnstr()</code> |
| | Change Renditions | | | | |
| <code>[mv][w]chgat()</code> | Change renditions of characters in a window | — | N | N | <code>chgat()</code> |
| | Delete | | | | |
| <code>[mv][w]delch()</code> | Delete a character | — | — | N | <code>delch()</code> |
| | Get (Input from Keyboard to Window) | | | | |
| <code>[mv][w]getch()</code> | Get a character | Y | Y | Y | <code>getch()</code> |
| <code>[mv][w]get[n]str()</code> | Get a character string | Y | Y | Y | <code>getnstr()</code> |
| <code>[mv][w]get_wch()</code> | Get a wide character | Y | Y | Y | <code>get_wch()</code> |
| <code>[mv][w]get[n]_wstr()</code> | Get an array of wide characters and key codes | Y | Y | Y | <code>get_wstr()</code> |
| | Explicit Cursor Movement | | | | |
| <code>[w]move()</code> | move the cursor | — | — | — | <code>move()</code> |
| | Input (Read Back from Window) | | | | |
| <code>[mv][w]inch()</code> | Input a character | — | — | — | <code>inch()</code> |
| <code>[mv][w]inch[n]str()</code> | Input an array of characters and attributes | — | — | — | <code>inchnstr()</code> |
| <code>[mv][w]in[n]str()</code> | Input a string | — | — | — | <code>innstr()</code> |

| Function Names | Description | s | w | c | Refer to |
|-----------------------|--|---|---|---|-------------|
| [mv][w]in[n]wstr() | Input a string of wide characters | — | — | — | innwstr() |
| [mv][w]in_wch() | Input a wide character and rendition | — | — | — | in_wch() |
| [mv][w]in_wch[n]str() | Input an array of wide characters and renditions | — | — | — | inwchnstr() |
| Insert | | | | | |
| [mv][w]insch() | Insert a character | Y | N | N | insch() |
| [mv][w]ins[n]str() | Insert a character string | Y | N | N | insnstr() |
| [mv][w]ins_[n]wstr() | Insert a wide-character string | Y | N | N | ins_nwstr() |
| [mv][w]ins_wch() | Insert a wide character | Y | N | N | ins_wch() |
| Print and Scan | | | | | |
| [mv][w]printw() | Print formatted output | — | — | — | moprintw() |
| [mv][w]scanw() | Convert formatted output | — | — | — | mvoscanw() |

Legend

The following notation indicates the effect when characters are moved to the screen. (For the Get functions, this applies only when echoing is enabled.)

- s Y means these functions perform special-character processing (see Section 3.4.3, on page 20). N means they do not. ? means the results are unspecified when these functions are applied to special characters.
- w Y means these functions perform wrapping (see Truncation, Wrapping, and Scrolling, on page 19). N means they do not.
- c Y means these functions advance the cursor (see Truncation, Wrapping, and Scrolling, on page 19). N means they do not.
- The attribute specified by this column does not apply to these functions.

3.7 Interfaces Implemented as Macros

The following interfaces with arguments shall be implemented as macros:

| | Macros | Reference Page |
|----|---|--------------------------------------|
| EC | COLOR_PAIR, PAIR_NUMBER() | Refer to <i>can_change_color()</i> . |
| EC | getbegyx(), getmaxyx(), getparyx(), getyx() | Refer to <i>getbegyx()</i> . |

EC The **int** arguments passed to *getbegyx()*, *getmaxyx()*, *getparyx()*, and *getyx()* shall be modifiable lvalues.

3.8 Initialized Curses Environment

Before executing an application that uses Curses, the terminal must be prepared as follows:

- If the terminal has hardware tab stops, they should be set.
- Any initialization strings defined for the terminal must be output to the terminal.

EC The resulting state of the terminal must be compatible with the model of the terminal that Curses has, as reflected in the terminal's entry in the **terminfo** database (see [Chapter 7](#)).

To initialize Curses, the application must call *initscr()* or *newterm()* before calling any of the other functions that deal with windows and screens, and it must call *endwin()* before exiting. To get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), the following sequence should be used:

```
initscr()
cbreak()
noecho()
```

Most programs would additionally use the sequence:

```
nonl()
intrflush(stdscr, FALSE)
keypad(stdscr, TRUE)
```

3.9 Synchronous and Networked Asynchronous Terminals

This section indicates to the application writer some considerations to be borne in mind when driving synchronous, networked asynchronous (NWA), or non-standard directly-connected asynchronous terminals.

Such terminals are often used in a mainframe environment and communicate to the host in block mode; that is, the user types characters at the terminal then presses a special key to initiate transmission of the characters to the host.

Frequently, although it may be possible to send arbitrary sized blocks to the host, it is not possible or desirable to cause a character to be transmitted with only a single keystroke.

This can cause severe problems to an application wishing to make use of single-character input; see [Section 3.5](#) (on page 22).

Output

The Curses interface can be used in the normal way for all operations pertaining to output to the terminal, with the possible exception that on some terminals the *refresh()* routine may have to redraw the entire screen contents in order to perform any update.

If it is additionally necessary to clear the screen before each such operation, the result could be undesirable.

Input

Because of the nature of operation of synchronous (block-mode) and NWA terminals, it might not be possible to support all or any of the Curses input functions. In particular, the following points should be noted:

- Single-character input might not be possible. It may be necessary to press a special key to cause all characters typed at the terminal to be transmitted to the host.
- It is sometimes not possible to disable echo. Character echo may be performed directly by the terminal. On terminals that behave in this way, any Curses application that performs input should be aware that any characters typed will appear on the screen at wherever the cursor is positioned. This does not necessarily correspond to the position of the cursor in the window.

Curses Interfaces

This chapter describes the Curses functions, macros, and external variables to support applications portability at the C-language source level.

The display model defined in [Section 3.4](#) (on page 18) contains important information, not repeated for individual interface definitions, regarding cursor movement, relocation of the cursor in the case of multi-column characters, wrapping of characters to subsequent lines of the screen, truncation of characters, and other important concepts. The reference pages must be read in conjunction with this overview information.

NAME

COLOR_PAIRS, COLORS — external variables for color support

SYNOPSIS

```
EC    #include <curses.h>
      extern int COLOR_PAIRS;
      extern int COLORS;
```

DESCRIPTION

Refer to [can_change_color\(\)](#).

NAME

COLS — number of columns on terminal screen

SYNOPSIS

```
EC #include <curses.h>
extern int COLS;
```

DESCRIPTION

The external variable *COLS* indicates the number of columns on the terminal screen.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

initscr(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

LINES — number of lines on terminal screen

SYNOPSIS

```
EC #include <curses.h>
extern int LINES;
```

DESCRIPTION

The external variable *LINES* indicates the number of lines on the terminal screen.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

initscr(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

add_wch, mvadd_wch, mvwadd_wch, wadd_wch — add a complex character and rendition to a window

SYNOPSIS

```
EC #include <curses.h>

int add_wch(const cchar_t *wch);
int mvadd_wch(int y, int x, const cchar_t *wch);
int mvwadd_wch(WINDOW *win, int y, int x, const cchar_t *wch);
int wadd_wch(WINDOW *win, const cchar_t *wch);
```

DESCRIPTION

These functions add information to the current or specified window at the current or specified position, and then advance the cursor. These functions perform special character processing. These functions perform wrapping.

- If *wch* refers to a spacing character, then any previous character at that location is removed, a new character specified by *wch* is placed at that location with rendition specified by *wch*; then the cursor advances to the next spacing character on the screen.
- If *wch* refers to a non-spacing character, all previous characters at that location are preserved, the non-spacing characters of *wch* are added to the spacing complex character, and the rendition specified by *wch* is ignored.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.4.4](#) (on page 21), [addch\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

add_wchnstr, add_wchstr, mvadd_wchnstr, mvadd_wchstr, mvwadd_wchnstr, mvwadd_wchstr, wadd_wchnstr, wadd_wchstr — add an array of complex characters and renditions to a window

SYNOPSIS

```
EC #include <curses.h>

int add_wchnstr(const cchar_t *wchstr, int n);
int add_wchstr(const cchar_t *wchstr);
int mvadd_wchnstr(int y, int x, const cchar_t *wchstr, int n);
int mvadd_wchstr(int y, int x, const cchar_t *wchstr);
int mvwadd_wchnstr(WINDOW *win, int y, int x, const cchar_t *wchstr,
int n);
int mvwadd_wchstr(WINDOW *win, int y, int x, const cchar_t *wchstr);
int wadd_wchnstr(WINDOW *win, const cchar_t *wchstr, int n);
int wadd_wchstr(WINDOW *win, const cchar_t *wchstr);
```

DESCRIPTION

These functions write the array of **cchar_t** specified by *wchstr* into the current or specified window starting at the current or specified cursor position.

These functions do not advance the cursor. The results are unspecified if *wchstr* contains any special characters.

These functions end successfully on encountering a null **cchar_t**. The functions also end successfully when they fill the current line. If a character cannot completely fit at the end of the current line, those columns are filled with the background character and rendition.

The *add_wchnstr()*, *mvadd_wchnstr()*, *mvwadd_wchnstr()*, and *wadd_wchnstr()* functions end successfully after writing *n* **cchar_ts** (or the entire array of **cchar_ts**, if *n* is -1).

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

addch, mvaddch, mvwaddch, waddch — add a single-byte character and rendition to a window and advance the cursor

SYNOPSIS

```
#include <curses.h>

int addch(const chtype ch);
int mvaddch(int y, int x, const chtype ch);
int mvwaddch(WINDOW *win, int y, int x, const chtype ch);
int waddch(WINDOW *win, const chtype ch);
```

DESCRIPTION

The *addch()*, *mvaddch()*, *mvwaddch()*, and *waddch()* functions place *ch* into the current or specified window at the current or specified position, and then advance the window's cursor position. These functions perform special character processing. These functions perform wrapping.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 3.4.4 (on page 21), *add_wch()*, *attroff()*, *doupdate()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity. Also the type of argument *ch* is changed from **chtype** to **const chtype**.

NAME

addchstr, addchnstr, mvaddchstr, mvaddchnstr, mvwaddchstr, mvwaddchnstr waddchstr, waddchnstr — add string of single-byte characters and renditions to a window

SYNOPSIS

```
#include <curses.h>

int addchstr(const chtype *chstr);
EC int addchnstr(const chtype *chstr, int n);
int mvaddchstr(int y, int x, const chtype *chstr);
EC int mvaddchnstr(int y, int x, const chtype *chstr, int n);
int mvwaddchstr(WINDOW *win, int y, int x, const chtype *chstr);
EC int mvwaddchnstr(WINDOW *win, int y, int x, const chtype *chstr,
int n);
int waddchstr(WINDOW *win, const chtype *chstr);
EC int waddchnstr(WINDOW *win, const chtype *chstr, int n);
```

DESCRIPTION

These functions overlay the contents of the current or specified window, starting at the current or specified position, with the contents of the array pointed to by *chstr* until a null **chtype** is encountered in the array pointed to by *chstr*.

These functions do not change the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

EC The *addchnstr()*, *mvaddchnstr()*, *mvwaddchnstr()*, and *waddchnstr()* functions copy at most *n* items, but no more than will fit on the current or specified line. If *n* is -1 then the whole string is copied, to the maximum number that fit on the current or specified line.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[addch\(\)](#), [add_wch\(\)](#), [add_wchnstr\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

addnstr, addstr, mvaddnstr, mvaddstr, mvwaddnstr, mvwaddstr, waddnstr, waddstr — add a string of multi-byte characters without rendition to a window and advance cursor

SYNOPSIS

```
EC #include <curses.h>

int addnstr(const char *str, int n);
int addstr(const char *str);
int mvaddnstr(int y, int x, const char *str, int n);
int mvaddstr(int y, int x, const char *str);
int mvwaddnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwaddstr(WINDOW *win, int y, int x, const char *str);
int waddnstr(WINDOW *win, const char *str, int n);
int waddstr(WINDOW *win, const char *str);
```

DESCRIPTION

These functions write the characters of the string *str* on the current or specified window starting at the current or specified position using the background rendition.

These functions advance the cursor position. These functions perform special character processing. These functions perform wrapping.

The *addstr()*, *mvaddstr()*, *mvwaddstr()*, and *waddstr()* functions are similar to calling *mbstowcs()* on *str*, and then calling *addwstr()*, *mvaddwstr()*, *mvwaddwstr()*, and *waddwstr()*, respectively.

The *addnstr()*, *mvaddnstr()*, *mvwaddnstr()*, and *waddnstr()* functions use at most *n* bytes from *str*. These functions add the entire string when *n* is -1 . These functions are similar to calling *mbstowcs()* on the first *n* bytes of *str*, and then calling *addwstr()*, *mvaddwstr()*, *mvwaddwstr()*, and *waddwstr()*, respectively.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

addnwstr(), *mbstowcs()* (in the XSH specification), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

In Issue 3, the *addstr()*, *mvaddstr()*, *mvwaddstr()*, and *waddstr()* functions were described in the *addstr()* entry. In Issue 4, the type of the *str* argument defined for these functions is changed from **char *** to **const char ***, and the DESCRIPTION was changed to indicate that the functions will handle multi-byte sequences correctly.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

addnwstr, addwstr, mvaddnwstr, mvaddwstr, mvwaddnwstr, mvwaddwstr, waddnwstr, waddwstr — add a wide-character string to a window and advance the cursor

SYNOPSIS

```
EC #include <curses.h>

int addnwstr(const wchar_t *wstr, int n);
int addwstr(const wchar_t *wstr);
int mvaddnwstr(int y, int x, const wchar_t *wstr, int n);
int mvaddwstr(int y, int x, const wchar_t *wstr);
int mvwaddnwstr(WINDOW *win, int y, int x, const wchar_t *wstr, int n);
int mvwaddwstr(WINDOW *win, int y, int x, const wchar_t *wstr);
int waddnwstr(WINDOW *win, const wchar_t *wstr, int n);
int waddwstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

These functions write the characters of the wide character string *wstr* on the current or specified window at that window's current or specified cursor position.

These functions advance the cursor position. These functions perform special character processing. These functions perform wrapping.

The effect is similar to building a `cchar_t` from the `wchar_t` and the background rendition and calling `wadd_wch()`, once for each `wchar_t` character in the string. The cursor movement specified by the *mv* functions occurs only once at the start of the operation.

The `addnwstr()`, `mvaddnwstr()`, `mvwaddnwstr()`, and `waddnwstr()` functions write at most *n* wide characters. If *n* is `-1`, then the entire string will be added.

RETURN VALUE

Upon successful completion, these functions return `OK`. Otherwise, they return `ERR`.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[add_wch\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

`attr_get`, `attr_off`, `attr_on`, `attr_set`, `color_set`, `wattr_get`, `wattr_off`, `wattr_on`, `wattr_set`, `wcolor_set` — window attribute control functions

SYNOPSIS

```
EC #include <curses.h>

int attr_get(attr_t *attrs, short *color_pair_number, void *opts);
int attr_off(attr_t attrs, void *opts);
int attr_on(attr_t attrs, void *opts);
int attr_set(attr_t attrs, short color_pair_number, void *opts);
int color_set(short color_pair_number, void *opts);
int wattr_get(WINDOW *win, attr_t *attrs, short *color_pair_number,
             void *opts);
int wattr_off(WINDOW *win, attr_t attrs, void *opts);
int wattr_on(WINDOW *win, attr_t attrs, void *opts);
int wattr_set(WINDOW *win, attr_t attrs, short color_pair_number,
             void *opts);
int wcolor_set(WINDOW *win, short color_pair_number, void *opts);
```

DESCRIPTION

These functions manipulate the attributes and color of the window rendition of the current or specified window.

The `attr_get()` and `wattr_get()` functions obtain the current rendition of a window. If `attrs` or `color_pair_number` is a null pointer, no information will be obtained on the corresponding rendition information and this is not an error.

The `attr_off()` and `wattr_off()` functions turn off `attrs` in the current or specified window without affecting any others.

The `attr_on()` and `wattr_on()` functions turn on `attrs` in the current or specified window without affecting any others.

The `attr_set()` and `wattr_set()` functions set the window rendition of the current or specified window to `attrs` and `color_pair_number`.

The `color_set()` and `wcolor_set()` functions set the window color of the current or specified window to `color_pair_number`.

RETURN VALUE

These functions always return OK.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

attroff(), **<curses.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

This entry is rewritten to include the color handling functions *wcolor_set()* and *color_set()*.

NAME

attroff, attron, attrset, wattroff, wattron, wattrset — restricted window attribute control functions

SYNOPSIS

```
#include <curses.h>

int attroff(int attrs);
int attron(int attrs);
int attrset(int attrs);
int wattroff(WINDOW *win, int attrs);
int wattron(WINDOW *win, int attrs);
int wattrset(WINDOW *win, int attrs);
```

DESCRIPTION

These functions manipulate the window attributes of the current or specified window.

The *attroff()* and *wattroff()* functions turn off *attrs* in the current or specified window without affecting any others.

The *attron()* and *wattron()* functions turn on *attrs* in the current or specified window without affecting any others.

The *attrset()* and *wattrset()* functions set the background attributes of the current or specified window to *attrs*.

It is unspecified whether these functions can be used to manipulate attributes other than *A_BLINK*, *A_BOLD*, *A_DIM*, *A_REVERSE*, *A_STANDOUT*, and *A_UNDERLINE*.

RETURN VALUE

These functions always return either OK or 1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Historical implementations returned either OK or 1. This revision allows either behavior.

FUTURE DIRECTIONS

None.

SEE ALSO

attr_get(), *standend()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

This entry is rewritten for clarity. The DESCRIPTION is updated to specify that it is undefined whether these functions can be used to manipulate attributes beyond those defined in Issue 3. The *standend()*, *standout()*, *wstandend()*, and *wstandout()* functions are moved to the *standend()* entry.

NAME

baudrate — get terminal baud rate

SYNOPSIS

```
#include <curses.h>

int baudrate(void);
```

DESCRIPTION

The *baudrate()* function extracts the output speed of the terminal in bits per second.

RETURN VALUE

The *baudrate()* function returns the output speed of the terminal.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcgetattr() (in the XSH specification), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The argument list is explicitly declared as **void**.

NAME

beep — audible signal

SYNOPSIS

```
#include <curses.h>

int beep(void);
```

DESCRIPTION

The *beep()* function alerts the user. It sounds the audible alarm on the terminal, or if that is not possible, it flashes the screen (visible bell). If neither signal is possible, nothing happens.

RETURN VALUE

The *beep()* function always returns OK.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Nearly all terminals have an audible alarm, but only some can flash the screen.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

flash(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The argument list is explicitly declared as **void**. The RETURN VALUE section is changed to indicate that the function always returns OK. The *flash()* function is moved to its own entry.

NAME

`bkgd`, `bkgdset`, `getbkgd`, `wbkgd`, `wbkgdset` — turn off the previous background attributes, logical OR the requested attributes into the window rendition, and set or get background character and rendition using a single-byte character

SYNOPSIS

```
EC #include <curses.h>
int bkgd(chtype ch);
void bkgdset(chtype ch);
chtype getbkgd(WINDOW *win);
int wbkgd(WINDOW *win, chtype ch);
void wbkgdset(WINDOW *win, chtype ch);
```

DESCRIPTION

The `bkgdset()` and `wbkgdset()` functions turn off the previous background attributes, logical OR the requested attributes into the window rendition, and set the background property of the current or specified window based on the information in *ch*. If *ch* refers to a multi-column character, the results are undefined.

The `bkgd()` and `wbkgd()` functions turn off the previous background attributes, logical OR the requested attributes into the window rendition, and set the background property of the current or specified window and then apply this setting to every character position in that window:

- The rendition of every character on the screen is changed to the new window rendition.
- Wherever the former background character appears, it is changed to the new background character.

The `getbkgd()` function extracts the specified window's background character and rendition.

RETURN VALUE

Upon successful completion, the `bkgd()` and `wbkgd()` functions return OK. Otherwise, they return ERR.

The `bkgdset()` and `wbkgdset()` functions do not return a value.

Upon successful completion, the `getbkgd()` function returns the specified window's background character and rendition. Otherwise, it returns (**chtype**)ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the `A_` prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.3.4](#) (on page 16), `<curses.h>`

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Rewritten for clarity.

NAME

bkgrnd, bkgrndset, getbkgrnd, wbkgrnd, wbkgrndset, wgetbkgrnd — turn off the previous background attributes, OR the requested attributes into the window rendition, and set or get background character and rendition using a complex character

SYNOPSIS

```
EC #include <curses.h>

int bkgrnd(const cchar_t *wch);
void bkgrndset(const cchar_t *wch);
int getbkgrnd(cchar_t *wch);
int wbkgrnd(WINDOW *win, const cchar_t *wch);
void wbkgrndset(WINDOW *win, const cchar_t *wch);
int wgetbkgrnd(WINDOW *win, cchar_t *wch);
```

DESCRIPTION

The *bkgrndset()* and *wbkgrndset()* functions turn off the previous background attributes, OR the requested attributes into the window rendition, and set the background property of the current or specified window based on the information in *wch*.

The *bkgrnd()* and *wbkgrnd()* functions turn off the previous background attributes, OR the requested attributes into the window rendition, and set the background property of the current or specified window and then apply this setting to every character position in that window:

- The rendition of every character on the screen is changed to the new window rendition.
- Wherever the former background character appears, it is changed to the new background character.

If *wch* refers to a non-spacing complex character for *bkgrnd()*, *bkgrndset()*, *wbkgrnd()*, and *wbkgrndset()*, then *wch* is added to the existing spacing complex character that is the background character. If *wch* refers to a multi-column character, the results are unspecified.

The *getbkgrnd()* and *wgetbkgrnd()* functions store, into the area pointed to by *wch*, the value of the window's background character and rendition.

RETURN VALUE

The *bkgrndset()* and *wbkgrndset()* functions do not return a value.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.3.4](#) (on page 16), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections applied.

NAME

border, wborder — draw borders from single-byte characters and renditions

SYNOPSIS

```
EC #include <curses.h>

int border(chtype ls, chtype rs, chtype ts, chtype bs, chtype tl,
           chtype tr, chtype bl, chtype br);
int wborder(WINDOW *win, chtype ls, chtype rs, chtype ts, chtype bs,
           chtype tl, chtype tr, chtype bl, chtype br);
```

DESCRIPTION

The *border()* and *wborder()* functions draw a border around the edges of the current or specified window. These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The arguments in the left-hand column of the following table contain single-byte characters with renditions, which have the following uses in drawing the border:

| Argument Name | Usage | Default Value |
|---------------|--|---------------|
| <i>ls</i> | Starting-column side | ACS_VLINE |
| <i>rs</i> | Ending-column side | ACS_VLINE |
| <i>ts</i> | First-line side | ACS_HLINE |
| <i>bs</i> | Last-line side | ACS_HLINE |
| <i>tl</i> | Corner of the first line and the starting column | ACS_ULCORNER |
| <i>tr</i> | Corner of the first line and the ending column | ACS_URCORNER |
| <i>bl</i> | Corner of the last line and the starting column | ACS_LLCORNER |
| <i>br</i> | Corner of the last line and the ending column | ACS_LRCORNER |

If the value of any argument in the left-hand column is 0, then the default value in the right-hand column is used. If the value of any argument in the left-hand column is a multi-column character, the results are undefined.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[border_set\(\)](#), [box\(\)](#), [hline\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 7

Corrigendum U022/2 is applied, changing the ACS_BLCORNER and ACS_BRCORNER macros to ACS_LLCORNER and ACS_LRCORNER, respectively.

NAME

border_set, wborder_set — draw borders from complex characters and renditions

SYNOPSIS

```
EC #include <curses.h>

int border_set(const cchar_t *ls, const cchar_t *rs, const cchar_t *ts,
               const cchar_t *bs, const cchar_t *tl, const cchar_t *tr,
               const cchar_t *bl, const cchar_t *br);
int wborder_set(WINDOW *win, const cchar_t *ls, const cchar_t *rs,
               const cchar_t *ts, const cchar_t *bs,
               const cchar_t *tl, const cchar_t *tr,
               const cchar_t *bl, const cchar_t *br);
```

DESCRIPTION

The *border_set()* and *wborder_set()* functions draw a border around the edges of the current or specified window. These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The arguments in the left-hand column of the following table contain spacing complex characters with renditions, which have the following uses in drawing the border:

| Argument Name | Usage | Default Value |
|---------------|--|---------------|
| <i>ls</i> | Starting-column side | WACS_VLINE |
| <i>rs</i> | Ending-column side | WACS_VLINE |
| <i>ts</i> | First-line side | WACS_HLINE |
| <i>bs</i> | Last-line side | WACS_HLINE |
| <i>tl</i> | Corner of the first line and the starting column | WACS_ULCORNER |
| <i>tr</i> | Corner of the first line and the ending column | WACS_URCORNER |
| <i>bl</i> | Corner of the last line and the starting column | WACS_LLCORNER |
| <i>br</i> | Corner of the last line and the ending column | WACS_LRCORNER |

If the value of any argument in the left-hand column is a null pointer, then the default value in the right-hand column is used. If the value of any argument in the left-hand column is a multi-column character, the results are undefined.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

box_set(), *hline_set()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

Issue 7

Corrigendum U022/2 is applied, changing the WACS_BLCORNER and WACS_BRCORNER macros to WACS_LLCORNER and WACS_LRCORNER, respectively.

NAME

box — draw borders from single-byte characters and renditions

SYNOPSIS

```
#include <curses.h>

int box(WINDOW *win, chtype verch, chtype horch);
```

DESCRIPTION

The *box()* function draws a border around the edges of the specified window. This function does not advance the cursor position. This function does not perform special character processing. This function does not perform wrapping.

The function *box(win, verch, horch)* has an effect equivalent to:

```
wborder(win, verch, verch, horch, horch, 0, 0, 0, 0);
```

RETURN VALUE

Upon successful completion, the *box()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This function is only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

border(), *box_set()*, *hline()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The DESCRIPTION is changed to describe this function in terms of a call to the *wborder()* function.

NAME

box_set — draw borders from complex characters and renditions

SYNOPSIS

```
EC #include <curses.h>
int box_set(WINDOW *win, const cchar_t *verch, const cchar_t *horch);
```

DESCRIPTION

The *box_set()* function draws a border around the edges of the specified window. This function does not advance the cursor position. This function does not perform special character processing. This function does not perform wrapping.

The function *box_set(win, verch, horch)* has an effect equivalent to:

```
wborder_set(win, verch, verch, horch, horch,
            NULL, NULL, NULL, NULL);
```

RETURN VALUE

Upon successful completion, the *box_set()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[border_set\(\)](#), [hline_set\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version

Corrections made to the SYNOPSIS.

NAME

can_change_color, color_content, has_colors, init_color, init_pair, pair_content, start_color — color manipulation functions

SYNOPSIS

```
EC #include <curses.h>

bool can_change_color(void);
int color_content(short color, short *red, short *green, short *blue);
int COLOR_PAIR(int n);
bool has_colors(void);
int init_color(short color, short red, short green, short blue);
int init_pair(short pair, short f, short b);
int pair_content(short pair, short *f, short *b);
int PAIR_NUMBER(int value);
int start_color(void);
extern int COLOR_PAIRS;
extern int COLORS;
```

DESCRIPTION

These functions manipulate color on terminals that support color.

Querying Capabilities

The *has_colors()* function indicates whether the terminal is a color terminal. The *can_change_color()* function indicates whether the terminal is a color terminal on which colors can be redefined.

Initialization

The *start_color()* function must be called in order to enable use of colors and before any color manipulation function is called. The function initializes eight basic colors (black, blue, green, cyan, red, magenta, yellow, and white) that can be specified by the color macros (such as `COLOR_BLACK`) defined in `<curses.h>` (see [Color-Related Macros](#), on page 309). The initial appearance of these eight colors is not specified.

The function also initializes two global external variables:

- `COLORS` defines the number of colors that the terminal supports (see [Color Identification](#)). If `COLORS` is 0, the terminal does not support redefinition of colors (and *can_change_color()* will return `FALSE`).
- `COLOR_PAIRS` defines the maximum number of color-pairs that the terminal supports (see [User-Defined Color Pairs](#), on page 57).

The *start_color()* function also restores the colors on the terminal to terminal-specific initial values. The initial background color is assumed to be black for all terminals.

Color Identification

The *init_color()* function redefines color number *color*, on terminals that support the redefinition of colors, to have the red, green, and blue intensity components specified by *red*, *green*, and *blue*, respectively. Calling *init_color()* also changes all occurrences of the specified color on the screen to the new definition.

The *color_content()* function identifies the intensity components of color number *color*. It stores the red, green, and blue intensity components of this color in the addresses pointed to by *red*, *green*, and *blue*, respectively.

For both functions, the *color* argument must be in the range from 0 to and including *COLORS*-1. Valid intensity values range from 0 (no intensity component) up to and including 1000 (maximum intensity in that component).

User-Defined Color Pairs

Calling *init_pair()* defines or redefines color-pair number *pair* to have foreground color *f* and background color *b*. Calling *init_pair()* changes any characters that were displayed in the color pair's old definition to the new definition and refreshes the screen.

After defining the color pair, the macro *COLOR_PAIR*(*n*) returns the value of color pair *n*. This value is the color attribute as it would be extracted from a **chtype**. Conversely, the macro *PAIR_NUMBER*(*value*) returns the color pair number associated with the color attribute *value*.

The *pair_content()* function retrieves the component colors of a color-pair number *pair*. It stores the foreground and background color numbers in the variables pointed to by *f* and *b*, respectively.

With *init_pair()* and *pair_content()*, the value of *pair* must be in a range from 0 to and including *COLOR_PAIRS*-1. (There may be an implementation-specific upper limit on the valid value of *pair*, but any such limit is at least 63.) Valid values for *f* and *b* are the range from 0 to and including *COLORS*-1.

RETURN VALUE

The *has_colors()* function returns TRUE if the terminal can manipulate colors. Otherwise, it returns FALSE.

The *can_change_color()* function returns TRUE if the terminal supports colors and can change their definitions. Otherwise, it returns FALSE.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To use these functions, *start_color()* must be called, usually right after *initscr()*.

The *can_change_color()* and *has_colors()* functions facilitate writing terminal-independent programs. For example, a programmer can use them to decide whether to use color or some other video attribute.

On color terminals, a typical value of *COLORS* is 8 and the macros such as *COLOR_BLACK* return a value within the range from 0 to and including 7. However, applications cannot rely on this to be true.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

attroff(), *delscreen()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version

Corrections made in the NAME and APPLICATION USAGE sections.

NAME

cbreak, nocbreak, noraw, raw — input mode control functions

SYNOPSIS

```
#include <curses.h>

int cbreak(void);
int nocbreak(void);
int noraw(void);
int raw(void);
```

DESCRIPTION

The *cbreak()* function sets the input mode for the current terminal to *cbreak* mode and overrides a call to *raw()*.

The *nocbreak()* function sets the input mode for the current terminal to Cooked Mode without changing the state of ISIG and IXON.

The *noraw()* function sets the input mode for the current terminal to Cooked Mode and sets the ISIG and IXON flags.

The *raw()* function sets the input mode for the current terminal to Raw Mode.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

If the application is not certain what the input mode of the process was at the time it called *initscr()*, it should use these functions to specify the desired input mode.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5.2](#) (on page 23), [<curses.h>](#), XBD specification, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 2.

Issue 4

The *raw()* and *noraw()* functions are merged with this entry.

The entry is rewritten for clarity.

The argument list for all these functions is explicitly declared as **void**.

NAME

chgat, mvchgat, mvwchgat, wchgat — change renditions of characters in a window

SYNOPSIS

```
EC #include <curses.h>

int chgat(int n, attr_t attr, short color, const void *opts);
int mvchgat(int y, int x, int n, attr_t attr, short color,
            const void *opts);
int mvwchgat(WINDOW *win, int y, int x, int n, attr_t attr,
            short color, const void *opts);
int wchgat(WINDOW *win, int n, attr_t attr, short color,
            const void *opts);
```

DESCRIPTION

These functions change the renditions of the next *n* characters in the current or specified window (or of the remaining characters on the current or specified line, if *n* is -1), starting at the current or specified cursor position. The attributes and colors are specified by *attr* and *color* as for *setcchar()*.

These functions do not update the cursor, except for the initial movement to the specified position by the functions prefixed with mv. These functions do not perform wrapping.

A value of *n* that is greater than the remaining characters on a line is not an error.

The *opts* argument is reserved for definition in a future version. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[setcchar\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

clear, erase, wclear, werase — clear a window

SYNOPSIS

```
#include <curses.h>

int clear(void);
int erase(void);
int wclear(WINDOW *win);
int werase(WINDOW *win);
```

DESCRIPTION

These functions clear every position in the current or specified window.

The *clear()* and *wclear()* functions also achieve the same effect as calling *clearok()*, so that the window is cleared completely on the next call to *wrefresh()* for the window and is redrawn in its entirety.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clearok(), *doupdate()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The *erase()* and *werase()* functions are merged with this entry.

The entry is rewritten for clarity.

The argument list for the *clear()* and *erase()* functions is explicitly declared as **void**.

NAME

clearok, idlok, leaveok, scrollok, setscrreg, wsetscrreg — terminal output control functions

SYNOPSIS

```
#include <curses.h>

int clearok(WINDOW *win, bool bf);
int idlok(WINDOW *win, bool bf);
int leaveok(WINDOW *win, bool bf);
int scrollok(WINDOW *win, bool bf);
int setscrreg(int top, int bot);
int wsetscrreg(WINDOW *win, int top, int bot);
```

DESCRIPTION

These functions set options that deal with output within Curses.

The *clearok()* function assigns the value of *bf* to an internal flag in the specified window that governs clearing of the screen during a refresh. If, during a refresh operation on the specified window, the flag in *curscr* is TRUE or the flag in the specified window is TRUE, then the implementation clears the screen, redraws it in its entirety, and sets the flag to FALSE in *curscr* and in the specified window. The initial state is unspecified.

The *idlok()* function specifies whether the implementation may use the hardware insert-line, delete-line, and scroll features of terminals so equipped. If *bf* is TRUE, use of these features is enabled. If *bf* is FALSE, use of these features is disabled and lines are instead redrawn as required. The initial state is FALSE.

The *leaveok()* function controls the cursor position after a refresh operation. If *bf* is TRUE, refresh operations on the specified window may leave the terminal's cursor at an arbitrary position. If *bf* is FALSE, then at the end of any refresh operation, the terminal's cursor is positioned at the cursor position contained in the specified window. The initial state is FALSE.

The *scrollok()* function controls the use of scrolling. If *bf* is TRUE, then scrolling is enabled for the specified window, with the consequences discussed in [Truncation, Wrapping, and Scrolling](#) (on page 19). If *bf* is FALSE, scrolling is disabled for the specified window. The initial state is FALSE.

The *setscrreg()* and *wsetscrreg()* functions define a software scrolling region in the current or specified window. The *top* and *bot* arguments are the line numbers of the first and last line defining the scrolling region. (Line 0 is the top line of the window.) If this option and *scrollok()* are enabled, an attempt to move off the last line of the margin causes all lines in the scrolling region to scroll one line in the direction of the first line. Only characters in the window are scrolled. If a software scrolling region is set and *scrollok()* is not enabled, an attempt to move off the last line of the margin does not reposition any lines in the scrolling region.

RETURN VALUE

Upon successful completion, the *setscrreg()* and *wsetscrreg()* functions return OK. Otherwise, they return ERR.

The other functions always return OK.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The only reason to enable the *idlok()* feature is to use scrolling to achieve the visual effect of motion of a partial window, such as for a screen editor. In other cases, the feature can be visually annoying.

The *leaveok()* option provides greater efficiency for applications that do not use the cursor.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clear(), *delscreen()*, *doupdate()*, *scl()*, <[curses.h](#)>

CHANGE HISTORY

First released in Issue 2.

Issue 4

The *idlok()*, *leaveok()*, *scrollok()*, *setscrreg()*, and *wsetscrreg()* functions are merged with this entry.

The entry is rewritten for clarity. The DESCRIPTION of *clearok()* is updated to indicate that clearing of a screen applies if the flag is TRUE in either *curscr* or the specified window.

The RETURN VALUE section is changed to indicate that the *clearok()*, *leaveok()*, and *scrollok()* functions always return OK.

NAME

clrrobot, wclrrobot — clear from cursor to end of window

SYNOPSIS

```
#include <curses.h>

int clrrobot(void);
int wclrrobot(WINDOW *win);
```

DESCRIPTION

These functions erase all lines following the cursor in the current or specified window, and erase the current line from the cursor to the end of the line, inclusive. These functions do not update the cursor.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[doupdate\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *clrrobot()* function is explicitly declared as **void**.

NAME

clrtoeol, wclrtoeol — clear from cursor to end of line

SYNOPSIS

```
#include <curses.h>

int clrtoeol(void);
int wclrtoeol(WINDOW *win);
```

DESCRIPTION

These functions erase the current line from the cursor to the end of the line, inclusive, in the current or specified window. These functions do not update the cursor.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

doupdate(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *clrtoeol()* function is explicitly declared as **void**.

NAME

color_content — identify red, green, and blue intensity of a color

SYNOPSIS

```
EC #include <curses.h>
    int color_content(short color, short *red, short *green, short *blue);
```

DESCRIPTION

Refer to [can_change_color\(\)](#).

NAME

color_set — window attribute control functions

SYNOPSIS

```
EC #include <curses.h>
    int color_set(short color_pair_number, void *opts);
```

DESCRIPTION

Refer to [attr_get\(\)](#).

NAME

copywin — copy a region of a window

SYNOPSIS

```
EC #include <curses.h>

int copywin(const WINDOW *srcwin, WINDOW *dstwin, int sminrow,
            int smincol, int dminrow, int dmincol, int dmaxrow,
            int dmaxcol, int overlay);
```

DESCRIPTION

The *copywin()* function provides a finer granularity of control over the *overlay()* and *overwrite()* functions. As in the *prefresh()* function, a rectangle is specified in the destination window (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window (*sminrow*, *smincol*). If *overlay* is TRUE, then copying is non-destructive, as in *overlay()*. If *overlay* is FALSE, then copying is destructive, as in *overwrite()*.

RETURN VALUE

Upon successful completion, this function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

newpad(), *overlay()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Corrections made to the SYNOPSIS.

NAME

cur_term — current terminal information

SYNOPSIS

```
EC #include <term.h>
extern TERMINAL *cur_term;
```

DESCRIPTION

The external variable *cur_term* identifies the record in the **terminfo** database associated with the terminal currently in use.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

del_curterm(), *tigetflag()*, **<term.h>**

CHANGE HISTORY

First released in Issue 4.

NAME

`curs_set` — set the cursor mode

SYNOPSIS

```
EC #include <curses.h>
int curs_set(int visibility);
```

DESCRIPTION

The `curs_set()` function sets the appearance of the cursor based on the value of *visibility*:

| Value of <i>visibility</i> | Appearance of Cursor |
|----------------------------|--|
| 0 | Invisible |
| 1 | Terminal-specific normal mode |
| 2 | Terminal-specific high visibility mode |

The terminal does not necessarily support all the above values.

RETURN VALUE

If the terminal supports the cursor mode specified by *visibility*, then the `curs_set()` function returns the previous cursor state. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

curscr — current window

SYNOPSIS

```
EC #include <curses.h>
extern WINDOW *curscr;
```

DESCRIPTION

The external variable *curscr* points to an internal data structure. It can be specified as an argument to certain functions, such as *clearok()*, where permitted in this specification.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clearok(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

def_prog_mode, def_shell_mode, reset_prog_mode, reset_shell_mode — save/restore program or shell terminal modes

SYNOPSIS

```
#include <curses.h>

int def_prog_mode(void);
int def_shell_mode(void);
int reset_prog_mode(void);
int reset_shell_mode(void);
```

DESCRIPTION

The *def_prog_mode()* function saves the current terminal modes as the “program” (in Curses) state for use by *reset_prog_mode()*.

The *def_shell_mode()* function saves the current terminal modes as the “shell” (not in Curses) state for use by *reset_shell_mode()*.

The *reset_prog_mode()* function restores the terminal to the “program” (in Curses) state.

The *reset_shell_mode()* function restores the terminal to the “shell” (not in Curses) state.

These functions affect the mode of the terminal associated with the current screen.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *initscr()* function achieves the effect of calling *def_shell_mode()* to save the prior terminal settings so they can be restored during the call to *endwin()*, and of calling *def_prog_mode()* to specify an initial definition of the program terminal mode.

Applications normally do not need to refer to the shell terminal mode. Applications may find it useful to save and restore the program terminal mode.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

doupdate(), *endwin()*, *initscr()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The *reset_prog_mode()* and *reset_shell_mode()* functions are merged with this entry.

The entry is rewritten for clarity.

The argument list for all these functions is explicitly declared as **void**.

NAME

del_curterm, restartterm, set_curterm, setupterm — interfaces to the **terminfo** database

SYNOPSIS

```
EC #include <term.h>

int del_curterm(TERMINAL *oterm);
int restartterm(char *term, int fildes, int *errret);
TERMINAL *set_curterm(TERMINAL *nterm);
int setupterm(char *term, int fildes, int *errret);
```

DESCRIPTION

These functions retrieve information from the **terminfo** database.

To gain access to the **terminfo** database, the *setupterm()* function must be called first. It is automatically called by *initscr()* and *newterm()*. The *setupterm()* function initializes the other functions to use the **terminfo** record for a specified terminal (which depends on whether *use_env()* was called). It sets the *cur_term* external variable to a **TERMINAL** structure that contains the record from the **terminfo** database for the specified terminal.

The terminal type is the character string *term*; if *term* is a null pointer, the environment variable *TERM* is used. If *TERM* is not set or if its value is an empty string, then **unknown** is used as the terminal type. The application must set *fildes* to a file descriptor, open for output, to the terminal device, before calling *setupterm()*. If *errret* is not null, the integer it points to is set to one of the following values to report the function outcome:

- 1 The **terminfo** database was not found (function fails).
- 0 The entry for the terminal was not found in **terminfo** (function fails).
- 1 Success.

If *setupterm()* detects an error and *errret* is a null pointer, the *setupterm()* function writes a diagnostic message and exits.

A simple call to *setupterm()* that uses all the defaults and sends the output to *stdout* is:

```
setupterm((char *)0, fileno(stdout), (int *)0);
```

The *set_curterm()* function sets the variable *cur_term* to *nterm*, and makes all of the **terminfo** boolean, numeric, and string variables use the values from *nterm*.

The *del_curterm()* function frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as *cur_term*, references to any of the **terminfo** boolean, numeric, and string variables thereafter may refer to invalid memory locations until *setupterm()* is called again.

The *restartterm()* function assumes a previous call to *setupterm()* (perhaps from *initscr()* or *newterm()*). It lets the application specify a different terminal type in *term* and updates the information returned by *baudrate()* based on *fildes*, but does not destroy other information created by *initscr()*, *newterm()*, or *setupterm()*.

RETURN VALUE

Upon successful completion, the *set_curterm()* function returns the previous value of *cur_term*. Otherwise, it returns a null pointer.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

An application would call *setupterm()* if it required access to the **terminfo** database but did not otherwise need to use Curses.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section A.3 (on page 377), *baudrate()*, *erasechar()*, *has_ic()*, *longname()*, *termattrs()*, *termname()*, *tigetflag()*, *use_env()*, **<term.h>**

CHANGE HISTORY

First released in Issue 4.

NAME

delay_output — delay output

SYNOPSIS

```
#include <curses.h>

int delay_output(int ms);
```

DESCRIPTION

On terminals that support pad characters, *delay_output()* pauses the output for at least *ms* milliseconds. Otherwise, the length of the delay is unspecified.

RETURN VALUE

Upon successful completion, the *delay_output()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Whether or not the terminal supports pad characters, the *delay_output()* function is not a precise method of timekeeping.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 7.1.3](#) (on page 340), *napms()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

NAME

delch, mvdelch, mvwdelch, wdelch — delete a character from a window

SYNOPSIS

```
#include <curses.h>

int delch(void);
int mvdelch(int y, int x);
int mvwdelch(WINDOW *win, int y, int x);
int wdelch(WINDOW *win);
```

DESCRIPTION

These functions delete the character at the current or specified position in the current or specified window. These functions do not change the cursor position.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *delch()* function is explicitly declared as **void**.

NAME

deleteln, wdeleteln — delete lines in a window

SYNOPSIS

```
#include <curses.h>

int deleteln(void);
int wdeleteln(WINDOW *win);
```

DESCRIPTION

These functions delete the line containing the cursor in the current or specified window and move all lines following the current line one line toward the cursor. The last line of the window is cleared. The cursor position does not change.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[insdelln\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *deleteln()* function is explicitly declared as **void**.

NAME

delscreen — free storage associated with a screen

SYNOPSIS

```
EC #include <curses.h>
void delscreen(SCREEN *sp);
```

DESCRIPTION

The *delscreen()* function frees storage associated with the **SCREEN** pointed to by *sp*.

RETURN VALUE

This function does not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endwin(), *initscr()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

delwin — delete a window

SYNOPSIS

```
#include <curses.h>

int delwin(WINDOW *win);
```

DESCRIPTION

This function deletes *win*, freeing all memory associated with it. The application must delete subwindows before deleting the main window.

RETURN VALUE

Upon successful completion, the *delwin()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

derwin(), *dupwin()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

NAME

derwin, newwin, subwin — window creation functions

SYNOPSIS

```
#include <curses.h>
```

```
EC WINDOW *derwin(WINDOW *orig, int nlines, int ncols, int begin_y,
                 int begin_x);
WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x);
WINDOW *subwin(WINDOW *orig, int nlines, int ncols, int begin_y,
              int begin_x);
```

DESCRIPTION

EC The *derwin()* function is the same as *subwin()*, except that *begin_y* and *begin_x* are relative to the origin of the window *orig* rather than absolute screen positions.

The *newwin()* function creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is (*begin_y*, *begin_x*). If *nlines* is zero, it defaults to *LINES* – *begin_y*; if *ncols* is zero, it defaults to *COLS* – *begin_x*. The size of a window cannot be greater than the physical size of the screen, or that defined using the environment variables *LINES* and *COLUMNS*. The behavior of a window which extends outside the terminal screen is undefined.

The *subwin()* function creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is at (*begin_y*, *begin_x*). (This position is an absolute screen position, not a position relative to the window *orig*.) If any part of the new window is outside *orig*, the function fails and the window is not created.

RETURN VALUE

Upon successful completion, these functions return a pointer to the new window. Otherwise, they return a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Before performing the first refresh of a subwindow, portable applications should call *touchwin()* or *touchline()* on the parent window.

Each window maintains internal descriptions of the screen image and status. The screen image is shared among all windows in the window hierarchy. Refresh operations rely on information on what has changed within a window, which is private to each window. Refreshing a window, when updates were made to a different window, may fail to perform needed updates because the windows do not share this information.

A new full-screen window is created by calling:

```
newwin(0, 0, 0, 0);
```

Pads should be used whenever a window larger than the terminal screen is required.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delwin(), *is_linetouched()*, *doupdate()*, [< curses.h >](#)

CHANGE HISTORY

First released in Issue 4.

Issue 7

Corrigendum U018/4 is applied, adding window size to the description of the *newwin()* function, and adding use of pads to the APPLICATION USAGE section.

NAME

doupdate, refresh, wnoutrefresh, wrefresh — refresh windows and lines

SYNOPSIS

```
#include <curses.h>

int doupdate(void);
int refresh(void);
int wnoutrefresh(WINDOW *win);
int wrefresh(WINDOW *win);
```

DESCRIPTION

The *refresh()* and *wrefresh()* functions refresh the current or specified window. The functions position the terminal's cursor at the cursor position of the window, except that if the *leaveok()* mode has been enabled, they may leave the cursor at an arbitrary position.

If the *win* parameter to *wrefresh()* is equal to the value of *curscr*, the screen is immediately cleared and repainted.

The *wnoutrefresh()* function determines which parts of the terminal may need updating. The *doupdate()* function sends to the terminal the commands to perform any required changes.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Refreshing an entire window is typically more efficient than refreshing several subwindows separately. An efficient sequence is to call *wnoutrefresh()* on each subwindow that has changed, followed by a call to *doupdate()*, which updates the terminal.

The *refresh()* or *wrefresh()* function (or *wnoutrefresh()* followed by *doupdate()*) must be called to send output to the terminal, as other Curses functions merely manipulate data structures.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[clearok\(\)](#), [curscr](#), [redrawwin\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

This entry is a merge of the Issue 3 entries *refresh()* and *wnoutrefresh()*. The DESCRIPTION is rewritten for clarity and the argument list for the *doupdate()* and *refresh()* functions is explicitly declared as **void**. Otherwise, the functionality is identical to that defined in Issue 3.

NAME

dupwin — duplicate a window

SYNOPSIS

```
EC #include <curses.h>
WINDOW *dupwin(WINDOW *win);
```

DESCRIPTION

This function creates a duplicate of the window *win*.

RETURN VALUE

Upon successful completion, the *dupwin()* function returns a pointer to the new window. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

derwin(), *doupdate()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

echo, noecho — enable/disable terminal echo

SYNOPSIS

```
#include <curses.h>

int echo(void);
int noecho(void);
```

DESCRIPTION

The *echo()* function enables Echo mode for the current screen. The *noecho()* function disables Echo mode for the current screen. Initially, curses software echo mode is enabled and hardware echo mode of the **tty** driver is disabled. *echo()* and *noecho()* control software echo only. Hardware echo must remain disabled for the duration of the application, else the behavior is undefined.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), *getch()*, [<curses.h>](#), XBD specification, Section 11.2, Parameters that Can be Set

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *echo()* and *noecho()* functions is explicitly declared as **void**.

Issue 4, Version 2

The state of the echo modes is further clarified.

NAME

echo_wchar, wecho_wchar — write a complex character and immediately refresh the window

SYNOPSIS

```
EC #include <curses.h>
int echo_wchar(const cchar_t *wch);
int wecho_wchar(WINDOW *win, const cchar_t *wch);
```

DESCRIPTION

The *echo_wchar()* function is equivalent to calling *add_wch()* and then calling *refresh()*.

The *wecho_wchar()* function is equivalent to calling *wadd_wch()* and then calling *wrefresh()*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

addch(), *add_wch()*, *doupdate()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

echochar, wechochar — echo single-byte character and rendition to a window and refresh

SYNOPSIS

```
EC #include <curses.h>
int echochar(const chtype ch);
int wechochar(WINDOW *win, const chtype ch);
```

DESCRIPTION

The *echochar()* function is equivalent to a call to *addch()* followed by a call to *refresh()*.

The *wechochar()* function is equivalent to a call to *waddch()* followed by a call to *wrefresh()*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[addch\(\)](#), [doupdate\(\)](#), [echo_wchar\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

endwin — suspend Curses session

SYNOPSIS

```
#include <curses.h>

int endwin(void);
```

DESCRIPTION

The *endwin()* function restores the terminal after Curses activity by at least restoring the saved shell terminal mode, flushing any output to the terminal, and moving the cursor to the first column of the last line of the screen. Refreshing a window resumes program mode. The application must call *endwin()* for each terminal being used before exiting. If *newterm()* is called more than once for the same terminal, the first screen created must be the last one for which *endwin()* is called.

RETURN VALUE

Upon successful completion, the *endwin()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *endwin()* function does not free storage associated with a screen, so *delscreen()* should be called after *endwin()* if a particular screen is no longer needed.

To leave Curses mode temporarily, portable applications should call *endwin()*. Subsequently, to return to Curses mode, they should call *doupdate()*, *refresh()*, or *wrefresh()*.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delscreen(), *doupdate()*, *initscr()*, *isendwin()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list is explicitly declared as **void**.

NAME

erase, werase — clear a window

SYNOPSIS

```
#include <curses.h>

int erase(void);
int werase(WINDOW *win);
```

DESCRIPTION

Refer to *clear()*.

NAME

erasechar, eraseswchar, killchar, killwchar — terminal environment query functions

SYNOPSIS

```
#include <curses.h>

char erasechar(void);
EC int eraseswchar(wchar_t *ch);
char killchar(void);
EC int killwchar(wchar_t *ch);
```

DESCRIPTION

EC The *erasechar()* function returns the current erase character. The *eraseswchar()* function stores the current erase character in the object pointed to by *ch*. If no erase character has been defined, the function will fail and the object pointed to by *ch* will not be changed.

EC The *killchar()* function returns the current line kill character. The *killwchar()* function stores the current line kill character in the object pointed to by *ch*. If no line kill character has been defined, the function will fail and the object pointed to by *ch* will not be changed.

RETURN VALUE

The *erasechar()* function returns the erase character and the *killchar()* function returns the line kill character. The return value is unspecified when these characters are multi-byte characters.

EC Upon successful completion, the *eraseswchar()* and *killwchar()* functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *erasechar()* and *killchar()* functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the *A_* prefix. Moreover, they do not reliably indicate cases in which when the erase or line kill character, respectively, has not been defined. The *eraseswchar()* and *killwchar()* functions overcome these limitations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 3.3.3 (on page 16), *clearok()*, *delscreen()*, *tcgetattr()* (in the XSH specification), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *erasechar()* and *killchar()* functions is explicitly declared as **void**.

The *erasewchar()* and *killwchar()* functions are added and marked as an X/Open UNIX Extension.

NAME

filter — disable use of certain terminal capabilities

SYNOPSIS

```
EC #include <curses.h>
void filter(void);
```

DESCRIPTION

The *filter()* function changes the algorithm for initializing terminal capabilities that assume that the terminal has more than one line. A subsequent call to *initscr()* or *newterm()* performs the following additional actions:

- Disable use of **clear**, **cu**, **cu1**, **cu**, **cu1**, and **vpa**.
- Set the value of the **home** string to the value of the **cr** string.
- Set **lines** equal to 1.

Any call to *filter()* must precede the call to *initscr()* or *newterm()*.

RETURN VALUE

The *filter()* function does not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 7.1.3](#) (on page 340), *initscr()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

flash — flash the screen

SYNOPSIS

```
#include <curses.h>

int flash(void);
```

DESCRIPTION

The *flash()* function alerts the user. It flashes the screen, or if that is not possible, it sounds the audible alarm on the terminal. If neither signal is possible, nothing happens.

RETURN VALUE

The *flash()* function always returns OK.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Nearly all terminals have an audible alarm, but only some can flash the screen.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

beep(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

In previous versions, this function was included in the entry for *beep()*. It is moved to its own entry in Issue 4, the argument list is explicitly declared as **void**, and the RETURN VALUE section is changed to indicate that the function always returns OK.

NAME

flushinp — discard input

SYNOPSIS

```
#include <curses.h>

int flushinp(void);
```

DESCRIPTION

The *flushinp()* function discards (flushes) any characters in the input buffer associated with the current screen.

RETURN VALUE

The *flushinp()* function always returns OK.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *flushinp()* function is explicitly declared as **void**.

NAME

get_wch, mvget_wch, mvwget_wch, wget_wch — get a wide character from a terminal

SYNOPSIS

```
EC #include <curses.h>

int get_wch(wint_t *ch);
int mvget_wch(int y, int x, wint_t *ch);
int mvwget_wch(WINDOW *win, int y, int x, wint_t *ch);
int wget_wch(WINDOW *win, wint_t *ch);
```

DESCRIPTION

These functions read a character from the terminal associated with the current or specified window. If *keypad()* is enabled, these functions respond to the pressing of a function key by setting the object pointed to by *ch* to the corresponding *KEY_* value defined in *<curses.h>* and returning *KEY_CODE_YES*.

Processing of terminal input is subject to the general rules described in [Section 3.5](#) (on page 22).

If echoing is enabled, then the character is echoed as though it were provided as an input argument to *add_wch()*, except for the following characters:

<backspace>, <left-arrow>, and the current erase character

The input is interpreted as specified in [Section 3.4.3](#) (on page 20) and then the character at the resulting cursor position is deleted as though *delch()* were called, except that if the cursor was originally in the first column of the line, then the user is alerted as though *beep()* were called.

Function keys

The user is alerted as though *beep()* were called. Information concerning the function keys is not returned to the caller.

If the current or specified window is not a pad, and it has been moved or modified since the last refresh operation, then it will be refreshed before another character is read.

RETURN VALUE

When these functions successfully report the pressing of a function key, they return *KEY_CODE_YES*. When they successfully report a wide character, they return *OK*. Otherwise, they return *ERR*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications should not define the escape key by itself as a single-character function.

When using these functions, nocbreak mode (*nocbreak()*) and echo mode (*echo()*) should not be used at the same time. Depending on the state of the terminal when each character is typed, the application may produce undesirable results.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), [beep\(\)](#), [cbreak\(\)](#), [ins_wch\(\)](#), [Section A.1.8](#), [move\(\)](#), [<urses.h>](#), [<wchar.h>](#)
(in the XBD specification)

CHANGE HISTORY

First released in Issue 4.

NAME

get_wstr — get an array of wide characters and function key codes from a terminal

SYNOPSIS

```
EC #include <curses.h>
    int get_wstr(wint_t *wstr);
```

DESCRIPTION

Refer to *getn_wstr()*.

NAME

getbegyx, getmaxyx, getparyx, getyx — get cursor and window coordinates

SYNOPSIS

```
#include <curses.h>
```

```
EC void getbegyx(WINDOW *win, int y, int x);
void getmaxyx(WINDOW *win, int y, int x);
void getparyx(WINDOW *win, int y, int x);
void getyx(WINDOW *win, int y, int x);
```

DESCRIPTION

The *getyx()* macro stores the cursor position of the specified window in *y* and *x*.

EC The *getparyx()* macro, if the specified window is a subwindow, stores in *y* and *x* the coordinates of the window's origin relative to its parent window. Otherwise, *-1* is stored in *y* and *x*.

The *getbegyx()* macro stores the absolute screen coordinates of the specified window's origin in *y* and *x*.

The *getmaxyx()* macro stores the number of rows of the specified window in *y* and stores the window's number of columns in *x*.

The application shall ensure that the *y* and *x* arguments are modifiable lvalues.

RETURN VALUE

No return values are defined.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Historical implementations often defined the following functions (which may have been implemented as macros):

```
int getbegx(WINDOW *win);
int getbegy(WINDOW *win);
int getcurx(WINDOW *win);
int getcury(WINDOW *win);
int getmaxx(WINDOW *win);
int getmaxy(WINDOW *win);
int getparx(WINDOW *win);
int getpary(WINDOW *win);
```

Although *getbegyx()*, *getyx()*, *getmaxyx()*, and *getparyx()* provide the required functionality, this does not preclude applications from defining these functions for their own use. For example, to implement:

```
int getbegx(WINDOW *win);
```

a suitable function would be:

```
int getbegx(WINDOW *win)
{
    int x, y;
    getbegyx(win, y, x);
    return x;
}
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the APPLICATION USAGE section.

NAME

getbkgd — get background character and rendition using a single-byte character

SYNOPSIS

```
EC #include <curses.h>
    chtype getbkgd(WINDOW *win);
```

DESCRIPTION

Refer to *bkgd()*.

NAME

getbkgrnd — get background character and rendition

SYNOPSIS

```
EC #include <curses.h>
   int getbkgrnd(cchar_t *ch);
```

DESCRIPTION

Refer to *bkgrnd()*.

NAME

getcchar — get a wide-character string and rendition from a **cchar_t**

SYNOPSIS

```
EC #include <curses.h>

int getcchar(const cchar_t *wcv, wchar_t *wch, attr_t *attrs,
             short *color_pair, void *opts);
```

DESCRIPTION

When *wch* is not a null pointer, the *getcchar()* function extracts information from a **cchar_t** defined by *wcv*, stores the character attributes in the object pointed to by *attrs*, stores the color pair in the object pointed to by *color_pair*, and stores the wide-character string referenced by *wcv* into the array pointed to by *wch*.

When *wch* is a null pointer, *getcchar()* obtains the number of wide characters in the object pointed to by *wcv* and does not change the objects pointed to by *attrs* or *color_pair*.

The *opts* argument is reserved for definition in a future version. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

When *wch* is a null pointer, the *getcchar()* function returns the number of wide characters referenced by *wcv*, including the null terminator.

When *wch* is not a null pointer, the *getcchar()* function returns OK upon successful completion. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *wcv* argument may be a value generated by a call to *setcchar()* or by a function that has a **cchar_t** output argument. If *wcv* is constructed by any other means, the effect is unspecified.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[attroff\(\)](#), [can_change_color\(\)](#), [setcchar\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

getch, mvgetch, mvwgetch, wgetch — get a single-byte character from the terminal

SYNOPSIS

```
#include < curses.h>

int getch(void);
int mvgetch(int y, int x);
int mvwgetch(WINDOW *win, int y, int x);
int wgetch(WINDOW *win);
```

DESCRIPTION

These functions read a single-byte character from the terminal associated with the current or specified window. The results are unspecified if the input is not a single-byte character. If *keypad()* is enabled, these functions respond to the pressing of a function key by returning the corresponding KEY_ value defined in **<curses.h>**.

Processing of terminal input is subject to the general rules described in [Section 3.5](#) (on page 22).

If echoing is enabled, then the character is echoed as though it were provided as an input argument to *addch()*, except for the following characters:

<backspace>, <left-arrow>, and the current erase character

The input is interpreted as specified in [Section 3.4.3](#) (on page 20) and then the character at the resulting cursor position is deleted as though *delch()* were called, except that if the cursor was originally in the first column of the line, then the user is alerted as though *beep()* were called.

Function keys

The user is alerted as though *beep()* were called. Information concerning the function keys is not returned to the caller.

If the current or specified window is not a pad, and it has been moved or modified since the last refresh operation, then it will be refreshed before another character is read.

RETURN VALUE

Upon successful completion, these functions return the single-byte character, KEY_ value, or ERR. When in the nodelay mode (*nodelay()*) and no data is available, ERR is returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications should not define the escape key by itself as a single-character function.

When using these functions, nocbreak mode (*nocbreak()*) and echo mode (*echo()*) should not be used at the same time. Depending on the state of the terminal when each character is typed, the program may produce undesirable results.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), *cbreak()*, *douupdate()*, *insch()*, `< curses.h >`

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *getch()* function is explicitly declared as **void**.

Issue 4, Version 2

The RETURN VALUE section is expanded.

NAME

getmaxyx — get size of a window

SYNOPSIS

```
EC #include <curses.h>
void getmaxyx(WINDOW *win, int y, int x);
```

DESCRIPTION

Refer to *getbegyx()*.

NAME

getn_wstr, get_wstr, mvgetn_wstr, mvget_wstr, mvwgetn_wstr, mvwget_wstr, wgetn_wstr, wget_wstr — get an array of wide characters and function key codes from a terminal

SYNOPSIS

```
EC #include <curses.h>

int getn_wstr(wint_t *wstr, int n);
int get_wstr(wint_t *wstr);
int mvgetn_wstr(int y, int x, wint_t *wstr, int n);
int mvget_wstr(int y, int x, wint_t *wstr);
int mvwgetn_wstr(WINDOW *win, int y, int x, wint_t *wstr, int n);
int mvwget_wstr(WINDOW *win, int y, int x, wint_t *wstr);
int wgetn_wstr(WINDOW *win, wint_t *wstr, int n);
int wget_wstr(WINDOW *win, wint_t *wstr);
```

DESCRIPTION

The effect of *get_wstr()* is as though a series of calls to *get_wch()* were made, until a <newline> character, end-of-line character, or end-of-file character is processed. An end-of-file character is represented by WEOF, as defined in <wchar.h>. A <newline> or end-of-line is represented as its **wchar_t** value. In all instances, the end of the string is terminated by a null **wchar_t**. The resulting values are placed in the area pointed to by *wstr*.

The user's erase and kill characters are interpreted and affect the sequence of characters returned.

The effect of *wget_wstr()* is as though a series of calls to *wget_wch()* were made.

The effect of *mvget_wstr()* is as though a call to *move()* followed by a series of calls to *get_wch()* were made. The effect of *mvwget_wstr()* is as though a call to *wmove()* followed by a series of calls to *wget_wch()* were made. The effect of *mvget_nwstr()* is as though a call to *move()* followed by a series of calls to *get_wch()* were made. The effect of *mvwget_nwstr()* is as though a call to *wmove()* followed by a series of calls to *wget_wch()* were made.

The *getn_wstr()*, *mvgetn_wstr()*, *mvwgetn_wstr()*, and *wgetn_wstr()* functions read at most *n* characters, letting the application prevent overflow of the input buffer.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *wstr* with *get_wstr()*, *mvget_wstr()*, *mvwget_wstr()*, or *wget_wstr()* causes undefined results. The use of *getn_wstr()*, *mvgetn_wstr()*, *mvwgetn_wstr()*, or *wgetn_wstr()*, respectively, is recommended.

These functions cannot return KEY_ values as there is no way to distinguish a KEY_ value from a valid **wchar_t** value.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[get_wch\(\)](#), [getnstr\(\)](#), [< curses.h >](#), [< wchar.h >](#) (in the XBD specification), XBD specification, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 4.

Issue 7

Corrigendum U018/1 is applied, correcting the [getn_wstr\(\)](#) and [get_wstr\(\)](#) function prototypes.

NAME

getnstr, getstr, mvgetnstr, mvgetstr, mvwgetnstr, mvwgetstr, wgetnstr, wgetstr — get a multi-byte character string from the terminal

SYNOPSIS

```
#include <curses.h>

EC   int getnstr(char *str, int n);
      int getstr(char *str);
EC   int mvgetnstr(int y, int x, char *str, int n);
      int mvgetstr(int y, int x, char *str);
EC   int mvwgetnstr(WINDOW *win, int y, int x, char *str, int n);
      int mvwgetstr(WINDOW *win, int y, int x, char *str);
EC   int wgetnstr(WINDOW *win, char *str, int n);
      int wgetstr(WINDOW *win, char *str);
```

DESCRIPTION

The effect of *getstr()* is as though a series of calls to *getch()* were made, until a <newline>, <carriage-return>, or end-of-file is received. The resulting value is placed in the area pointed to by *str*. The string is then terminated with a null byte. The *getnstr()*, *mvgetnstr()*, *mvwgetnstr()*, and *wgetnstr()* functions are equivalent to the *getstr()*, *mvgetstr()*, *mvwgetstr()*, and *wgetstr()* functions respectively, except that they read at most *n*-1 bytes, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, home key, clear key, and so on).

The *mvgetstr()* function is identical to *getstr()* except that it is as though it is a call to *move()* followed by a series of calls to *getch()*. The *mvwgetstr()* function is identical to *getstr()* except it is as though a call to *wmove()* is made followed by a series of calls to *wgetch()*. The *mvgetnstr()* function is identical to *getnstr()* except that it is as though it is a call to *move()* followed by a series of calls to *getch()*. The *mvwgetnstr()* function is identical to *getnstr()* except it is as though a call to *wmove()* is made followed by a series of calls to *wgetch()*.

The *getnstr()*, *wgetnstr()*, *mvgetnstr()*, and *mvwgetnstr()* functions will only return the entire multi-byte sequence associated with a character. If the array is large enough to contain at least one character, the functions fill the array with complete characters. If the array is not large enough to contain any complete characters, the function fails.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *str* with *getstr()*, *mvgetstr()*, *mvwgetstr()*, or *wgetstr()* causes undefined results. The use of *getnstr()*, *mvgetnstr()*, *mvwgetnstr()*, or *wgetnstr()*, respectively, is recommended.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 3.5 (on page 22), *beep()*, *getch()*, **< curses.h >**

CHANGE HISTORY

First released in Issue 4.

In Issue 3, the *getstr()*, *mvgetstr()*, *mvwgetstr()*, and *wgetstr()* functions were described in the *addstr()* entry. In Issue 4, the DESCRIPTION of these functions is rewritten for clarity and is updated to indicate that they will handle multi-byte sequences correctly.

Issue 4, Version 2

Corrections made to first sentence of the DESCRIPTION.

NAME

getparyx — get subwindow origin coordinates

SYNOPSIS

```
EC #include <curses.h>
    void getparyx(WINDOW *win, int y, int x);
```

DESCRIPTION

Refer to *getbegyx()*.

NAME

getstr — get a multi-byte character string from the terminal

SYNOPSIS

```
#include <curses.h>
int getstr(char *str);
```

DESCRIPTION

Refer to *getnstr()*.

NAME

getwin, putwin — dump window to, and reload window from, a file

SYNOPSIS

```
EC #include <curses.h>
WINDOW *getwin(FILE *filep);
int putwin(WINDOW *win, FILE *filep);
```

DESCRIPTION

The *getwin()* function reads window-related data stored in the file by *putwin()*. The function then creates and initializes a new window using that data.

The *putwin()* function writes all data associated with *win* into the *stdio* stream to which *filep* points, using an unspecified format. This information can be retrieved later using *getwin()*.

RETURN VALUE

Upon successful completion, the *getwin()* function returns a pointer to the window it created. Otherwise, it returns a null pointer.

Upon successful completion, the *putwin()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

scr_dump(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

getyx — get cursor coordinates

SYNOPSIS

```
#include <curses.h>

void getyx(WINDOW *win, int y, int x);
```

DESCRIPTION

Refer to *getbegyx()*.

NAME

halfdelay — control input character delay mode

SYNOPSIS

```
EC #include <curses.h>
   int halfdelay(int tenths);
```

DESCRIPTION

The *halfdelay()* function sets the input mode for the current window to Half-Delay Mode and specifies *tenths* tenths of seconds as the half-delay interval. The *tenths* argument must be in a range from 1 up to and including 255.

RETURN VALUE

Upon successful completion, the *halfdelay()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The application can call *nocbreak()* to leave Half-Delay mode.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 3.5.2 (on page 23), *cbreak()*, [<curses.h>](#), XBD specification, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 4.

NAME

has_colors — indicate whether terminal supports colors

SYNOPSIS

```
EC #include <curses.h>
    bool has_colors(void);
```

DESCRIPTION

Refer to [can_change_color\(\)](#).

NAME

has_ic, has_il — query functions for terminal insert and delete capability

SYNOPSIS

```
#include <curses.h>

bool has_ic(void);
bool has_il(void);
```

DESCRIPTION

The *has_ic()* function indicates whether the terminal has insert-character and delete-character capabilities.

The *has_il()* function indicates whether the terminal has insert-line and delete-line capabilities, or can simulate them using scrolling regions.

RETURN VALUE

The *has_ic()* function returns TRUE if the terminal has insert-character and delete-character capabilities. Otherwise, it returns FALSE.

The *has_il()* function returns TRUE if the terminal has insert-line and delete-line capabilities. Otherwise, it returns FALSE.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *has_il()* function may be used to determine whether it would be appropriate to turn on physical scrolling using *scrollok()*.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The *has_il()* function is merged with this entry.

The entry is rewritten for clarity.

The argument list for the *has_ic()* and *has_il()* functions is explicitly declared as **void**.

NAME

hline, *mvhline*, *mvvline*, *mvwhline*, *mvwvline*, *vline*, *whline*, *wvline* — draw lines from single-byte characters and renditions

SYNOPSIS

```
EC #include <curses.h>

int hline(chtype ch, int n);
int mvhline(int y, int x, chtype ch, int n);
int mvvline(int y, int x, chtype ch, int n);
int mvwhline(WINDOW *win, int y, int x, chtype ch, int n);
int mvwvline(WINDOW *win, int y, int x, chtype ch, int n);
int vline(chtype ch, int n);
int whline(WINDOW *win, chtype ch, int n);
int wvline(WINDOW *win, chtype ch, int n);
```

DESCRIPTION

These functions draw a line in the current or specified window starting at the current or specified position, using *ch*. The line is at most *n* positions long, or as many as fit into the window.

These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The *hline()*, *mvhline()*, *mvwhline()*, and *whline()* functions draw a line proceeding toward the last column of the same line.

The *vline()*, *mvvline()*, *mvwvline()*, and *wvline()* functions draw a line proceeding toward the last line of the window.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

border(), *box()*, *hline_set()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

`hline_set`, `mvhline_set`, `mvvline_set`, `mvwhline_set`, `mvwvline_set`, `vline_set`, `whline_set`, `wvline_set` — draw lines from complex characters and renditions

SYNOPSIS

```
EC #include <curses.h>

int hline_set(const cchar_t *wch, int n);
int mvhline_set(int y, int x, const cchar_t *wch, int n);
int mvvline_set(int y, int x, const cchar_t *wch, int n);
int mvwhline_set(WINDOW *win, int y, int x, const cchar_t *wch, int n);
int mvwvline_set(WINDOW *win, int y, int x, const cchar_t *wch, int n);
int vline_set(const cchar_t *wch, int n);
int whline_set(WINDOW *win, const cchar_t *wch, int n);
int wvline_set(WINDOW *win, const cchar_t *wch, int n);
```

DESCRIPTION

These functions draw a line in the current or specified window starting at the current or specified position, using *ch*. The line is at most *n* positions long, or as many as fit into the window.

These functions do not advance the cursor position. These functions do not perform special character processing. These functions do not perform wrapping.

The `hline_set()`, `mvhline_set()`, `mvwhline_set()`, and `whline_set()` functions draw a line proceeding toward the last column of the same line.

The `vline_set()`, `mvvline_set()`, `mvwvline_set()`, and `wvline_set()` functions draw a line proceeding toward the last line of the window.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[*border_set\(\)*](#), [**<curses.h>**](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

idcok — enable or disable use of hardware insert-character and delete-character features

SYNOPSIS

```
EC #include <curses.h>
void idcok(WINDOW *win, bool bf);
```

DESCRIPTION

The *idcok()* function specifies whether the implementation may use hardware insert-character and delete-character features in *win* if the terminal is so equipped. If *bf* is TRUE, use of these features in *win* is enabled. If *bf* is FALSE, use of these features in *win* is disabled. The initial state is TRUE.

RETURN VALUE

The *idcok()* function does not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clearok(), *doupdate()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

idlok — enable or disable use of terminal insert-character and delete-line features

SYNOPSIS

```
#include <curses.h>

int idlok(WINDOW *win, bool bf);
```

DESCRIPTION

Refer to *clearok()*.

NAME

immedok — enable or disable immediate terminal refresh

SYNOPSIS

```
EC #include <curses.h>
void immedok(WINDOW *win, bool bf);
```

DESCRIPTION

The *immedok()* function specifies whether the screen is refreshed whenever the window pointed to by *win* is changed. If *bf* is TRUE, the window is implicitly refreshed on each such change. If *bf* is FALSE, the window is not implicitly refreshed. The initial state is FALSE.

RETURN VALUE

The *immedok()* function does not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *immedok()* function is useful for windows that are used as terminal emulators.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[clearok\(\)](#), [doupdate\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

in_wch, mvin_wch, mvwin_wch, win_wch — extract a complex character and rendition from a window

SYNOPSIS

```
EC #include <curses.h>

int in_wch(cchar_t *wcval);
int mvin_wch(int y, int x, cchar_t *wcval);
int mvwin_wch(WINDOW *win, int y, int x, cchar_t *wcval);
int win_wch(WINDOW *win, cchar_t *wcval);
```

DESCRIPTION

These functions extract the complex character and rendition from the current or specified position in the current or specified window into the object pointed to by *wcval*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

`in_wchnstr`, `in_wchstr`, `mvin_wchnstr`, `mvin_wchstr`, `mvwin_wchnstr`, `mvwin_wchstr`, `win_wchnstr`, `win_wchstr` — extract an array of complex characters and renditions from a window

SYNOPSIS

```
EC #include <curses.h>

int in_wchnstr(cchar_t *wchstr, int n);
int in_wchstr(cchar_t *wchstr);
int mvin_wchnstr(int y, int x, cchar_t *wchstr, int n);
int mvin_wchstr(int y, int x, cchar_t *wchstr);
int mvwin_wchnstr(WINDOW *win, int y, int x, cchar_t *wchstr, int n);
int mvwin_wchstr(WINDOW *win, int y, int x, cchar_t *wchstr);
int win_wchnstr(WINDOW *win, cchar_t *wchstr, int n);
int win_wchstr(WINDOW *win, cchar_t *wchstr);
```

DESCRIPTION

These functions extract characters from the current or specified window, starting at the current or specified position and ending at the end of the line, and place them in the array pointed to by *wchstr*.

The `in_wchnstr()`, `mvin_wchnstr()`, `mvwin_wchnstr()`, and `win_wchnstr()` functions fill the array with at most *n* **cchar_t** elements.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *wchstr* with `in_wchstr()`, `mvin_wchstr()`, `mvwin_wchstr()`, or `win_wchstr()` causes undefined results. The use of `in_wchnstr()`, `mvin_wchnstr()`, `mvwin_wchnstr()`, or `win_wchnstr()`, respectively, is recommended.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[*in_wch\(\)*](#), [**<curses.h>**](#)

CHANGE HISTORY

First released in Issue 4.

NAME

inch, mvinch, mvwinch, winch — input a single-byte character and rendition from a window

SYNOPSIS

```
#include <curses.h>

chtype inch(void);
chtype mvinch(int y, int x);
chtype mvwinch(WINDOW *win, int y, int x);
chtype winch(WINDOW *win);
```

DESCRIPTION

These functions return the character and rendition, of type *chtype*, at the current or specified position in the current or specified window.

RETURN VALUE

Upon successful completion, the functions return the specified character and rendition. Otherwise, they return (**chtype**)ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *inch()* function is explicitly declared as **void**.

NAME

inchnstr, inchstr, mvinchnstr, mvinchstr, mvwinchnstr, mvwinchstr, winchnstr, winchstr — input an array of single-byte characters and renditions from a window

SYNOPSIS

```
EC #include <curses.h>

int inchnstr(chtype *chstr, int n);
int inchstr(chtype *chstr);
int mvinchnstr(int y, int x, chtype *chstr, int n);
int mvinchstr(int y, int x, chtype *chstr);
int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);
int winchnstr(WINDOW *win, chtype *chstr, int n);
int winchstr(WINDOW *win, chtype *chstr);
```

DESCRIPTION

These functions place characters and renditions from the current or specified window into the array pointed to by *chstr*, starting at the current or specified position and ending at the end of the line.

The *inchnstr()*, *mvinchnstr()*, *mvwinchnstr()*, and *winchnstr()* functions store at most *n* elements from the current or specified window into the array pointed to by *chstr*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *chstr* with *inchstr()*, *mvinchstr()*, *mvwinchstr()*, or *winchstr()* causes undefined results. The use of *inchnstr()*, *mvinchnstr()*, *mvwinchnstr()*, or *winchnstr()*, respectively, is recommended.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

inch(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

init_color, init_pair — redefine specified color or color pair

SYNOPSIS

```
EC    #include <curses.h>
      int init_color(short color, short red, short green, short blue);
      int init_pair(short pair, short f, short b);
```

DESCRIPTION

Refer to [can_change_color\(\)](#).

NAME

initscr, newterm — screen initialization functions

SYNOPSIS

```
#include <curses.h>

WINDOW *initscr(void);
SCREEN *newterm(const char *type, FILE *outfile, FILE *infile);
```

DESCRIPTION

The *initscr()* function determines the terminal type and initializes all implementation data structures. The *TERM* environment variable specifies the terminal type. The *initscr()* function also causes the first refresh operation to clear the screen. If errors occur, *initscr()* writes an appropriate error message to standard error and exits. The only functions that can be called before *initscr()* or *newterm()* are *filter()*, *ripoffline()*, *slk_init()*, *use_env()*, and the functions whose prototypes are defined in **<term.h>**. Portable applications must not call *initscr()* twice.

The *newterm()* function can be called as many times as desired to attach a terminal device. The *type* argument points to a string specifying the terminal type, except that if *type* is a null pointer, the *TERM* environment variable is used. The *outfile* and *infile* arguments are file pointers for output to the terminal and input from the terminal, respectively. It is unspecified whether Curses modifies the buffering mode of these file pointers. The *newterm()* function should be called once for each terminal.

The *initscr()* function is equivalent to:

```
newterm(getenv("TERM"), stdout, stdin);
return stdscr;
```

If the current disposition for the signals SIGINT, SIGQUIT, or SIGTSTP is SIGDFL, then *initscr()* may also install a handler for the signal, which may remain in effect for the life of the process or until the process changes the disposition of the signal.

The *initscr()* and *newterm()* functions initialize the *cur_term* external variable.

RETURN VALUE

Upon successful completion, the *initscr()* function returns a pointer to *stdscr*. Otherwise, it does not return.

Upon successful completion, the *newterm()* function returns a pointer to the specified terminal. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

A program that outputs to more than one terminal should use *newterm()* for each terminal instead of *initscr()*. A program that needs an indication of error conditions, so it can continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this function.

Applications should perform any required handling of the SIGINT, SIGQUIT, or SIGTSTP signals before calling *initscr()*.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section A.3 (on page 377), *delscreen()*, *doupdate()*, *del_curterm()*, *filter()*, *slk_atroff()*, *use_env()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The *newterm()* function is merged with this entry.

The entry is rewritten for clarity.

The argument list for the *initscr()* function is explicitly declared as **void**.

Issue 7

The prototype for the *newterm()* function is updated.

NAME

innstr, *instr*, *mvinnstr*, *mvinstr*, *mvwinnstr*, *mvwinstr*, *winnstr*, *winstr* — input a multi-byte character string from a window

SYNOPSIS

```
EC #include <curses.h>

int innstr(char *str, int n);
int instr(char *str);
int mvinnstr(int y, int x, char *str, int n);
int mvinstr(int y, int x, char *str);
int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
int mvwinstr(WINDOW *win, int y, int x, char *str);
int winnstr(WINDOW *win, char *str, int n);
int winstr(WINDOW *win, char *str);
```

DESCRIPTION

These functions place a string of characters from the current or specified window into the array pointed to by *str*, starting at the current or specified position and ending at the end of the line.

The *innstr()*, *mvinnstr()*, *mvwinnstr()*, and *winnstr()* functions store at most *n* bytes in the string pointed to by *str*.

The *innstr()*, *mvinnstr()*, *mvwinnstr()*, and *winnstr()* functions will only store the entire multi-byte sequence associated with a character. If the array is large enough to contain at least one character, the array is filled with complete characters. If the array is not large enough to contain any complete characters, the function fails.

RETURN VALUE

Upon successful completion, the *instr()*, *mvinstr()*, *mvwinstr()*, and *winstr()* functions return OK.

Upon successful completion, the *innstr()*, *mvinnstr()*, *mvwinnstr()*, and *winnstr()* functions return the number of characters actually read into the string.

Otherwise, all these functions return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Since multi-byte characters may be processed, there might not be a one-to-one correspondence between the number of column positions on the screen and the number of bytes returned.

These functions do not return rendition information.

Reading a line that overflows the array pointed to by *str* with *instr()*, *mvinstr()*, *mvwinstr()*, or *winstr()* causes undefined results. The use of *innstr()*, *mvinnstr()*, *mvwinnstr()*, or *winnstr()*, respectively, is recommended.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

innwstr, inwstr, mvinnwstr, mvinwstr, mvwinnwstr, mvwinwstr, winnwstr, winwstr — input a string of wide characters from a window

SYNOPSIS

```
EC #include <curses.h>

int innwstr(wchar_t *wstr, int n);
int inwstr(wchar_t *wstr);
int mvinnwstr(int y, int x, wchar_t *wstr, int n);
int mvinwstr(int y, int x, wchar_t *wstr);
int mvwinnwstr(WINDOW *win, int y, int x, wchar_t *wstr, int n);
int mvwinwstr(WINDOW *win, int y, int x, wchar_t *wstr);
int winnwstr(WINDOW *win, wchar_t *wstr, int n);
int winwstr(WINDOW *win, wchar_t *wstr);
```

DESCRIPTION

These functions place a string of **wchar_t** characters from the current or specified window into the array pointed to by *wstr* starting at the current or specified cursor position and ending at the end of the line.

These functions will only store the entire wide-character sequence associated with a spacing complex character. If the array is large enough to contain at least one complete spacing complex character, the array is filled with complete characters. If the array is not large enough to contain any complete characters, this is an error.

The *innwstr()*, *mvinnwstr()*, *mvwinnwstr()*, and *winnwstr()* functions store at most *n* characters in the array pointed to by *wstr*.

RETURN VALUE

Upon successful completion, the *inwstr()*, *mvinwstr()*, *mvwinwstr()*, and *winwstr()* functions return OK.

Upon successful completion, the *innwstr()*, *mvinnwstr()*, *mvwinnwstr()*, and *winnwstr()* functions return the number of characters actually read into the string.

Otherwise, all these functions return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *wstr* with *inwstr()*, *mvinwstr()*, *mvwinwstr()*, or *winwstr()* causes undefined results. The use of *innwstr()*, *mvinnwstr()*, *mvwinnwstr()*, or *winnwstr()*, respectively, is recommended.

These functions do not return rendition information.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

ins_nwstr, ins_wstr, mvins_nwstr, mvins_wstr, mvwins_nwstr, mvwins_wstr, wins_nwstr, wins_wstr — insert a wide-character string into a window

SYNOPSIS

```
EC #include <curses.h>

int ins_nwstr(const wchar_t *wstr, int n);
int ins_wstr(const wchar_t *wstr);
int mvins_nwstr(int y, int x, const wchar_t *wstr, int n);
int mvins_wstr(int y, int x, const wchar_t *wstr);
int mvwins_nwstr(WINDOW *win, int y, int x, const wchar_t *wstr,
                int n);
int mvwins_wstr(WINDOW *win, int y, int x, const wchar_t *wstr);
int wins_nwstr(WINDOW *win, const wchar_t *wstr, int n);
int wins_wstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

These functions insert a **wchar_t** character string (as many **wchar_t** characters as will fit on the line) in the current or specified window immediately before the current or specified position.

Any non-spacing characters in the string are associated with the first spacing character in the string that precedes the non-spacing characters. If the first character in the string is a non-spacing character, these functions will fail.

These functions do not advance the cursor position. These functions perform special character processing. These functions do not perform wrapping.

The *ins_nwstr()*, *mvins_nwstr()*, *mvwins_nwstr()*, and *wins_nwstr()* functions insert at most *n* **wchar_t** characters. If *n* is less than 0, then the entire string is inserted.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

ins_wch, mvins_wch, mvwins_wch, wins_wch — insert a complex character and rendition into a window

SYNOPSIS

```
EC #include <curses.h>

int ins_wch(const cchar_t *wch);
int mvins_wch(int y, int x, const cchar_t *wch);
int mvwins_wch(WINDOW *win, int y, int x, const cchar_t *wch);
int wins_wch(WINDOW *win, const cchar_t *wch);
```

DESCRIPTION

These functions insert the complex character *wch* with its rendition in the current or specified window at the current or specified cursor position.

These functions do not advance the cursor position. These functions perform special-character processing, with the exception that if a <newline> is inserted into the last line of a window and scrolling is not enabled, the behavior is unspecified. These functions do not perform wrapping.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For non-spacing characters, *add_wch()* can be used to add the non-spacing characters to a spacing complex character already in the window.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[add_wch\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

`insch`, `mvinsch`, `mvwinsch`, `winsch` — insert a single-byte character and rendition into a window

SYNOPSIS

```
#include <curses.h>

int insch(chtype ch);
int mvinsch(int y, int x, chtype ch);
int mvwinsch(WINDOW *win, int y, int x, chtype ch);
int winsch(WINDOW *win, chtype ch);
```

DESCRIPTION

These functions insert the character and rendition from *ch* into the current or specified window at the current or specified position.

These functions do not advance the cursor position. These functions perform special character processing, with the exception that if a <newline> is inserted into the last line of a window and scrolling is not enabled, the behavior is unspecified. These functions do not perform wrapping.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ins_wch(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

Issue 4, Version 2

The DESCRIPTION is further clarified.

NAME

insdelln, winsdelln — delete or insert lines into a window

SYNOPSIS

```
EC #include <curses.h>
int insdelln(int n);
int winsdelln(WINDOW *win, int n);
```

DESCRIPTION

These functions perform the following actions:

- If *n* is positive, these functions insert *n* lines into the current or specified window before the current line. The *n* last lines are no longer displayed.
- If *n* is negative, these functions delete *n* lines from the current or specified window starting with the current line, and move the remaining lines toward the cursor. The last *n* lines are cleared.

The current cursor position remains the same.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[deleteln\(\)](#), [insertln\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

insertln, winsertln — insert lines into a window

SYNOPSIS

```
#include <curses.h>

int insertln(void);
int winsertln(WINDOW *win);
```

DESCRIPTION

These functions insert a blank line before the current line in the current or specified window. The bottom line is no longer displayed. The cursor position does not change.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[insdelln\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *insertln()* function is explicitly declared as **void**.

NAME

insnstr, insstr, mvinsnstr, mvinsstr, mvwinsnstr, mvwinsstr, winsnstr, winsstr — insert a multi-byte character string into a window

SYNOPSIS

```
EC #include <curses.h>

int insnstr(const char *str, int n);
int insstr(const char *str);
int mvinsnstr(int y, int x, const char *str, int n);
int mvinsstr(int y, int x, const char *str);
int mvwinsnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwinsstr(WINDOW *win, int y, int x, const char *str);
int winsnstr(WINDOW *win, const char *str, int n);
int winsstr(WINDOW *win, const char *str);
```

DESCRIPTION

These functions insert a character string (as many characters as will fit on the line) before the current or specified position in the current or specified window.

These functions do not advance the cursor position. These functions perform special character processing. These functions do not perform wrapping.

The *insnstr()*, *mvinsnstr()*, *mvwinsnstr()*, and *winsnstr()* functions insert at most *n* bytes. If *n* is less than 1, the entire string is inserted.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Since the string may contain multi-byte characters, there might not be a one-to-one correspondence between the number of column positions occupied by the characters and the number of bytes in the string.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

instr — input a multi-byte character string from the current window

SYNOPSIS

```
EC #include <curses.h>
   int instr(char *str);
```

DESCRIPTION

Refer to *innstr()*.

NAME

intrflush — enable or disable flush on interrupt

SYNOPSIS

```
#include <curses.h>

int intrflush(WINDOW *win, bool bf);
```

DESCRIPTION

The *intrflush()* function specifies whether pressing an interrupt key (interrupt, suspend, or quit) will flush the input buffer associated with the current screen. If the value of *bf* is TRUE, then flushing of the output buffer associated with the current screen will occur when an interrupt key (interrupt, suspend, or quit) is pressed. If the value of *bf* is FALSE, then no flushing of the buffer will occur when an interrupt key is pressed. The default for the option is inherited from the display driver settings. The *win* argument is ignored.

RETURN VALUE

Upon successful completion, the *intrflush()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The same effect is achieved outside Curses using the NOFLSH local mode flag specified in the XBD specification (**General Terminal Interface**).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), [<curses.h>](#), XBD specification, Section 11.2, Parameters that Can be Set

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

Issue 4, Version 2

The description of the *bf* argument has been changed to align with Issue 3 and preserve compatibility.

NAME

inwstr — input a string of wide characters from the current window

SYNOPSIS

```
EC #include <curses.h>
   int inwstr(wchar_t *wstr);
```

DESCRIPTION

Refer to *innwstr()*.

NAME

is_linetouched, is_wintouched, touchline, touchwin, untouchwin, wtouchln — window refresh control functions

SYNOPSIS

```
#include <curses.h>
```

```
EC bool is_linetouched(WINDOW *win, int line);
bool is_wintouched(WINDOW *win);
int touchline(WINDOW *win, int start, int count);
int touchwin(WINDOW *win);
EC int untouchwin(WINDOW *win);
int wtouchln(WINDOW *win, int y, int n, int changed);
```

DESCRIPTION

EC The *touchwin()* function touches the specified window (that is, marks it as having changed more recently than the last refresh operation). The *touchline()* function only touches *count* lines, beginning with line *start*.

The *untouchwin()* function marks all lines in the window as unchanged since the last refresh operation.

Calling *wtouchln()*, if *changed* is 1, touches *n* lines in the specified window, starting at line *y*. If *changed* is 0, *wtouchln()* marks such lines as unchanged since the last refresh operation.

The *is_wintouched()* function determines whether the specified window is touched. The *is_linetouched()* function determines whether line *line* of the specified window is touched.

RETURN VALUE

EC The *is_linetouched()* and *is_wintouched()* functions return TRUE if any of the specified lines, or the specified window, respectively, has been touched since the last refresh operation. Otherwise, they return FALSE.

Upon successful completion, the other functions return OK. Otherwise, they return ERR. Exceptions to this are noted in the preceding function descriptions.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Calling *touchwin()* or *touchline()* is sometimes necessary when using overlapping windows, since a change to one window affects the other window, but the records of which lines have been changed in the other window do not reflect the change.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.2](#) (on page 14), *doupdate()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

isendwin — determine whether a screen has been refreshed

SYNOPSIS

```
EC #include <curses.h>
    bool isendwin(void);
```

DESCRIPTION

The *isendwin()* function indicates whether the screen has been refreshed since the last call to *endwin()*.

RETURN VALUE

The *isendwin()* function returns TRUE if *endwin()* has been called without any subsequent refresh. Otherwise, it returns FALSE.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endwin(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

keyname, key_name — get name of key

SYNOPSIS

```
EC #include <curses.h>
char *keyname(int c);
char *key_name(wchar_t c);
```

DESCRIPTION

These functions generate a character string whose value describes the key *c*. The *c* argument of *keyname()* can be an 8-bit character or a key code. The *c* argument of *key_name()* must be a wide character.

The string has a format according to the first applicable row in the following table:

| Input | Format of Returned String |
|--|---------------------------|
| Visible character | The same character |
| Control character | ^X |
| Meta-character (<i>keyname()</i> only) | -X |
| Key value defined in <curses.h> (<i>keyname()</i> only) | KEY_name |
| None of the above | UNKNOWN KEY |

The meta-character notation shown above is used only if meta-characters are enabled.

RETURN VALUE

Upon successful completion, these functions return a pointer to a string as described above. Otherwise, they return a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The return value of *keyname()* and *key_name()* may point to a static area which is overwritten by a subsequent call to either of these functions.

Applications normally process meta-characters without storing them into a window. If an application stores meta-characters in a window and tries to retrieve them as wide characters, *keyname()* cannot detect meta-characters, since wide characters do not support meta-characters.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

meta(), **<curses.h>**

CHANGE HISTORY

First released in Issue 4.

NAME

keypad — enable/disable abbreviation of function keys

SYNOPSIS

```
#include <curses.h>

int keypad(WINDOW *win, bool bf);
```

DESCRIPTION

The *keypad()* function controls keypad translation. If *bf* is TRUE, keypad translation is turned on. If *bf* is FALSE, keypad translation is turned off. The initial state is FALSE.

This function affects the behavior of any function that provides keyboard input.

If the terminal in use requires a command to enable it to transmit distinctive codes when a function key is pressed, then after keypad translation is first enabled, the implementation transmits this command to the terminal before an affected input function tries to read any characters from that terminal.

RETURN VALUE

Upon successful completion, the *keypad()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5.1](#) (on page 22), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

NAME

killchar, killwchar — terminal environment query functions

SYNOPSIS

```
#include <curses.h>
char killchar(void);
EC int killwchar(wchar_t *ch);
```

DESCRIPTION

Refer to *erasechar()*.

NAME

leaveok — control cursor position resulting from refresh operations

SYNOPSIS

```
#include <curses.h>

int leaveok(WINDOW *win, bool bf);
```

DESCRIPTION

Refer to *clearok()*.

NAME

leaveok — terminal output control functions

SYNOPSIS

```
#include <curses.h>

int leaveok(WINDOW *win, bool bf);
```

DESCRIPTION

Refer to *clearok()*.

NAME

longname — get verbose description of current terminal

SYNOPSIS

```
#include <curses.h>

char *longname(void);
```

DESCRIPTION

The *longname()* function generates a verbose description of the current terminal. The maximum length of a verbose description is 128 bytes. It is defined only after the call to *initscr()* or *newterm()*.

RETURN VALUE

Upon successful completion, the *longname()* function returns a pointer to the description specified above. Otherwise, it returns a null pointer on error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The return value of *longname()* may point to a static area which is overwritten by a subsequent call to *newterm()*.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

initscr(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *longname()* function is explicitly declared as **void**.

NAME

meta — enable/disable meta-keys

SYNOPSIS

```
EC #include <curses.h>
int meta(WINDOW *win, bool bf);
```

DESCRIPTION

Initially, whether the terminal returns seven or eight significant bits on input depends on the control mode of the display driver (see the **XBD** specification, **General Terminal Interface**). To force eight bits to be returned, invoke *meta(win, TRUE)*. To force seven bits to be returned, invoke *meta(win, FALSE)*. The *win* argument is always ignored. If the **terminfo** capabilities **smm** (*meta_on*) and **rmm** (*meta_off*) are defined for the terminal, **smm** is sent to the terminal when *meta(win, TRUE)* is called and **rmm** is sent when *meta(win, FALSE)* is called.

RETURN VALUE

Upon successful completion, the *meta()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The same effect is achieved outside Curses using the CS7 or CS8 control mode flag specified in the **XBD** specification (**General Terminal Interface**).

The *meta()* function was designed for use with terminals with 7-bit character sets and a “meta” key that could be used to set the eighth bit.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), *getch()*, [<curses.h>](#), **XBD** specification, Section 11.2, Parameters that Can be Set (ISTRIP flag)

CHANGE HISTORY

First released in Issue 4.

NAME

move, wmove — window cursor location functions

SYNOPSIS

```
#include <curses.h>

int move(int y, int x);
int wmove(WINDOW *win, int y, int x);
```

DESCRIPTION

These functions move the cursor associated with the current or specified window to (y, x) relative to the window's origin. This function does not move the terminal's cursor until the next refresh operation.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[doupdate\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

NAME

mv — pointer page for functions with mv prefix

DESCRIPTION

Most cases in which a Curses function has the mv prefix indicate that the function takes *y* and *x* arguments and moves the cursor to that address as though *move()* were first called. (The corresponding functions without the mv prefix operate at the cursor position.)

Note: The *mvcur()*, *moverwin()*, and *mvwin()* functions are exceptions to this rule, in that mv is not a prefix with the usual meaning and there are no corresponding functions without the mv prefix. These functions have entries under their own names.

In the *mvprintw()* and *mvscanw()* functions, mv is a prefix with the usual meaning, but the functions have entries under their own names because the *mv()* function is the first function in the family of functions in alphabetical order.

The mv prefix is combined with a w prefix to produce Curses functions beginning with mvw.

The *mv*()* and *mvw*()* functions are discussed together with the corresponding functions that do not have these prefixes. They are found on the following entries:

| Function | Refer to |
|--|----------------------|
| <i>mvaddch()</i> , <i>mvwaddch()</i> | <i>addch()</i> |
| <i>mvaddchnstr()</i> , <i>mvwaddchnstr()</i> | <i>addchnstr()</i> |
| <i>mvaddchstr()</i> , <i>mvwaddchstr()</i> | <i>addchstr()</i> |
| <i>mvaddnstr()</i> , <i>mvwaddnstr()</i> | <i>addnstr()</i> |
| <i>mvaddstr()</i> , <i>mvwaddstr()</i> | <i>addstr()</i> |
| <i>mvaddnwstr()</i> , <i>mvwaddnwstr()</i> | <i>addnwstr()</i> |
| <i>mvaddwstr()</i> , <i>mvwaddwstr()</i> | <i>addwstr()</i> |
| <i>mvadd_wch()</i> , <i>mvwadd_wch()</i> | <i>add_wch()</i> |
| <i>mvadd_wchnstr()</i> , <i>mvwadd_wchnstr()</i> | <i>add_wchnstr()</i> |
| <i>mvadd_wchstr()</i> , <i>mvwadd_wchstr()</i> | <i>add_wchstr()</i> |
| <i>mvchgat()</i> , <i>mvwchgat()</i> | <i>chgat()</i> |
| <i>mvdelch()</i> , <i>mvwdelch()</i> | <i>delch()</i> |
| <i>mvgetch()</i> , <i>mvwgetch()</i> | <i>getch()</i> |
| <i>mvgetnstr()</i> , <i>mvwgetnstr()</i> | <i>getnstr()</i> |
| <i>mvgetstr()</i> , <i>mvwgetstr()</i> | <i>getstr()</i> |
| <i>mvgetn_wstr()</i> , <i>mvwgetn_wstr()</i> | <i>getn_wstr()</i> |
| <i>mvget_wch()</i> , <i>mvwget_wch()</i> | <i>get_wch()</i> |
| <i>mvget_wstr()</i> , <i>mvwget_wstr()</i> | <i>getwstr()</i> |
| <i>mvhline()</i> , <i>mvwhline()</i> | <i>hline()</i> |
| <i>mvhline_set()</i> , <i>mvwhline_set()</i> | <i>hline_set()</i> |
| <i>mvinch()</i> , <i>mvwinch()</i> | <i>inch()</i> |
| <i>mvinchnstr()</i> , <i>mvwinchnstr()</i> | <i>inchnstr()</i> |
| <i>mvinchstr()</i> , <i>mvwinchstr()</i> | <i>inchstr()</i> |
| <i>mvinnstr()</i> , <i>mvwinnstr()</i> | <i>innstr()</i> |
| <i>mvinnwstr()</i> , <i>mvwinnwstr()</i> | <i>innwstr()</i> |
| <i>mvinsch()</i> , <i>mvwinsch()</i> | <i>insch()</i> |
| <i>mvinsnstr()</i> , <i>mvwinsnstr()</i> | <i>insnstr()</i> |
| <i>mvinsstr()</i> , <i>mvwinsstr()</i> | <i>insstr()</i> |
| <i>mvinstr()</i> , <i>mvwinstr()</i> | <i>instr()</i> |
| <i>mvins_wstr()</i> , <i>mvwins_wstr()</i> | <i>ins_wstr()</i> |
| <i>mvins_wch()</i> , <i>mvwins_wch()</i> | <i>ins_wch()</i> |
| <i>mvins_wstr()</i> , <i>mvwins_wstr()</i> | <i>ins_wstr()</i> |
| <i>mvinwstr()</i> , <i>mvwinwstr()</i> | <i>inwstr()</i> |

| Function | Refer to |
|---|---------------------|
| <i>mvin_wch()</i> , <i>mrowin_wch()</i> | <i>in_wch()</i> |
| <i>mvin_wchnstr()</i> , <i>mrowin_wchnstr()</i> | <i>in_wchnstr()</i> |
| <i>mvin_wchstr()</i> , <i>mrowin_wchstr()</i> | <i>in_wchstr()</i> |
| <i>moprintw()</i> , <i>mrowprintw()</i> | <i>moprintw()</i> |
| <i>mvoscanw()</i> , <i>mrowscanw()</i> | <i>mvoscanw()</i> |
| <i>mvvline()</i> , <i>mrowvline()</i> | <i>hline()</i> |
| <i>mvvline_set()</i> , <i>mrowvline_set()</i> | <i>hline_set()</i> |

NAME

mvadd_wch, mvwadd_wch — add a complex character and rendition to a window

SYNOPSIS

```
EC #include <curses.h>
int mvadd_wch(int y, int x, const cchar_t *wch);
int mvwadd_wch(WINDOW *win, int y, int x, const cchar_t *wch);
```

DESCRIPTION

Refer to [add_wch\(\)](#).

NAME

mvadd_wchnstr, mvadd_wchstr, mvwadd_wchnstr, mvwadd_wchstr — add an array of complex characters and renditions to a window

SYNOPSIS

```
EC #include <curses.h>

int mvadd_wchnstr(int y, int x, const cchar_t *wchstr, int n);
int mvadd_wchstr(int y, int x, const cchar_t *wchstr);
int mvwadd_wchnstr(WINDOW *win, int y, int x, const cchar_t *wchstr,
int n);
int mvwadd_wchstr(WINDOW *win, int y, int x, const cchar_t *wchstr);
```

DESCRIPTION

Refer to [add_wchnstr\(\)](#).

NAME

mvaddch, mvwaddch — add a single-byte character and rendition to a window and advance the cursor

SYNOPSIS

```
#include <curses.h>

int mvaddch(int y, int x, const chtype ch);
int mvwaddch(WINDOW *win, int y, int x, const chtype ch);
```

DESCRIPTION

Refer to *addch()*.

NAME

mvaddchstr, mvaddchnstr, mvwaddchstr, mvwaddchnstr — add string of single-byte characters and renditions to a window

SYNOPSIS

```
#include <curses.h>

int mvaddchstr(int y, int x, const chtype *chstr);
EC int mvaddchnstr(int y, int x, const chtype *chstr, int n);
int mvwaddchstr(WINDOW *win, int y, int x, const chtype *chstr);
EC int mvwaddchnstr(WINDOW *win, int y, int x, const chtype *chstr,
int n);
```

DESCRIPTION

Refer to [addchstr\(\)](#).

NAME

mvaddnstr, mvaddstr, mvwaddnstr, mvwaddstr — add a string of multi-byte characters without rendition to a window and advance cursor

SYNOPSIS

```
EC #include <curses.h>

int mvaddnstr(int y, int x, const char *str, int n);
int mvaddstr(int y, int x, const char *str);
int mvwaddnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwaddstr(WINDOW *win, int y, int x, const char *str);
```

DESCRIPTION

Refer to [addnstr\(\)](#).

NAME

mvaddnwstr, mvaddwstr, mvwaddnwstr, mvwaddwstr — add a wide-character string to a window and advance the cursor

SYNOPSIS

```
EC #include <curses.h>

int mvaddnwstr(int y, int x, const wchar_t *wstr, int n);
int mvaddwstr(int y, int x, const wchar_t *wstr);
int mvwaddnwstr(WINDOW *win, int y, int x, const wchar_t *wstr, int n);
int mvwaddwstr(WINDOW *win, int y, int x, const wchar_t *wstr);
```

DESCRIPTION

Refer to [addnwstr\(\)](#).

NAME

mvchgat, mvwchgat — change renditions of characters in a window

SYNOPSIS

```
EC #include <curses.h>

int mvchgat(int y, int x, int n, attr_t attr, short color,
            const void *opts);
int mvwchgat(WINDOW *win, int y, int x, int n, attr_t attr,
            short color, const void *opts);
```

DESCRIPTION

Refer to *chgat()*.

NAME

mvcur — output cursor movement commands to the terminal

SYNOPSIS

```
EC #include <curses.h>
int mvcur(int oldrow, int oldcol, int newrow, int newcol);
```

DESCRIPTION

The *mvcur()* function outputs one or more commands to the terminal that moves the terminal's cursor to (*newrow*, *newcol*), an absolute position on the terminal screen. The (*oldrow*, *oldcol*) arguments specify the former cursor position. Specifying the former position is necessary on terminals that do not provide coordinate-based movement commands. On terminals that provide these commands, Curses may select a more efficient way to move the cursor based on the former position. If (*newrow*, *newcol*) is not a valid address for the terminal in use, *mvcur()* fails. If (*oldrow*, *oldcol*) is the same as (*newrow*, *newcol*), then *mvcur()* succeeds without taking any action. If *mvcur()* outputs a cursor movement command, it updates its information concerning the location of the cursor on the terminal.

RETURN VALUE

Upon successful completion, the *mvcur()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

After use of *mvcur()*, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[doupdate\(\)](#), [is_linetouched\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

mvdelch, mvwdelch — delete a character from a window

SYNOPSIS

```
#include <curses.h>

int mvdelch(int y, int x);
int mvwdelch(WINDOW *win, int y, int x);
```

DESCRIPTION

Refer to *delch()*.

NAME

mvderwin — define window coordinate transformation

SYNOPSIS

```
EC #include <curses.h>
int mvderwin(WINDOW *win, int par_y, int par_x);
```

DESCRIPTION

The *mvderwin()* function specifies a mapping of characters. The function identifies a mapped area of the parent of the specified window, whose size is the same as the size of the specified window and whose origin is at (*par_y*, *par_x*) of the parent window.

- During any refresh of the specified window, the characters displayed in that window's display area of the terminal are taken from the mapped area.
- Any references to characters in the specified window obtain or modify characters in the mapped area.

That is, *mvderwin()* defines a coordinate transformation from each position in the mapped area to a corresponding position (same *y*, *x* offset from the origin) in the specified window.

RETURN VALUE

Upon successful completion, the *mvderwin()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[derwin\(\)](#), [doupdate\(\)](#), [dupwin\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

mvget_wch, mvwget_wch — get a wide character from a terminal

SYNOPSIS

```
EC #include <curses.h>
    int mvget_wch(int y, int x, wint_t *ch);
    int mvwget_wch(WINDOW *win, int y, int x, wint_t *ch);
```

DESCRIPTION

Refer to [get_wch\(\)](#).

NAME

mvgetch, mvwgetch — get a single-byte character from the terminal

SYNOPSIS

```
#include <curses.h>

int mvgetch(int y, int x);
int mvwgetch(WINDOW *win, int y, int x);
```

DESCRIPTION

Refer to [getch\(\)](#).

NAME

mvgetn_wstr, mvget_wstr, mvwgetn_wstr, mvwget_wstr — get an array of wide characters and function key codes from a terminal

SYNOPSIS

```
EC #include <curses.h>

int mvgetn_wstr(int y, int x, wint_t *wstr, int n);
int mvget_wstr(int y, int x, wint_t *wstr);
int mvwgetn_wstr(WINDOW *win, int y, int x, wint_t *wstr, int n);
int mvwget_wstr(WINDOW *win, int y, int x, wint_t *wstr);
```

DESCRIPTION

Refer to [getn_wstr\(\)](#).

NAME

mvgetnstr, mvgetstr, mvwgetnstr, mvwgetstr — get a multi-byte character string from the terminal

SYNOPSIS

```
#include <curses.h>
```

```
EC int mvgetnstr(int y, int x, char *str, int n);
```

```
int mvgetstr(int y, int x, char *str);
```

```
EC int mvwgetnstr(WINDOW *win, int y, int x, char *str, int n);
```

```
int mvwgetstr(WINDOW *win, int y, int x, char *str);
```

DESCRIPTION

Refer to [getnstr\(\)](#).

NAME

mvhline, mvvline, mvwhline, mvwvline — draw lines from single-byte characters and renditions

SYNOPSIS

```
EC #include <curses.h>

int mvhline(int y, int x, chtype ch, int n);
int mvvline(int y, int x, chtype ch, int n);
int mvwhline(WINDOW *win, int y, int x, chtype ch, int n);
int mvwvline(WINDOW *win, int y, int x, chtype ch, int n);
```

DESCRIPTION

Refer to *hline()*.

NAME

mvhline_set, mvvline_set, mvwhline_set, mvwvline_set — draw lines from complex characters and renditions

SYNOPSIS

```
EC #include <curses.h>

int mvhline_set(int y, int x, const cchar_t *wch, int n);
int mvvline_set(int y, int x, const cchar_t *wch, int n);
int mvwhline_set(WINDOW *win, int y, int x, const cchar_t *wch, int n);
int mvwvline_set(WINDOW *win, int y, int x, const cchar_t *wch, int n);
```

DESCRIPTION

Refer to [hline_set\(\)](#).

NAME

mvin_wch, mvwin_wch — extract a complex character and rendition from a window

SYNOPSIS

```
EC #include <curses.h>
int mvin_wch(int y, int x, cchar_t *wcval);
int mvwin_wch(WINDOW *win, int y, int x, cchar_t *wcval);
```

DESCRIPTION

Refer to [in_wch\(\)](#).

NAME

`mvin_wchnstr`, `mvin_wchstr`, `mvwin_wchnstr`, `mvwin_wchstr` — extract an array of complex characters and renditions from a window

SYNOPSIS

```
EC #include <curses.h>

int mvin_wchnstr(int y, int x, cchar_t *wchstr, int n);
int mvin_wchstr(int y, int x, cchar_t *wchstr);
int mvwin_wchnstr(WINDOW *win, int y, int x, cchar_t *wchstr, int n);
int mvwin_wchstr(WINDOW *win, int y, int x, cchar_t *wchstr);
```

DESCRIPTION

Refer to [in_wchnstr\(\)](#).

NAME

mvinch, mvwinch — input a single-byte character and rendition from a window

SYNOPSIS

```
#include <curses.h>

chtype mvinch(int y, int x);
chtype mvwinch(WINDOW *win, int y, int x);
```

DESCRIPTION

Refer to *inch()*.

NAME

mvinchstr, mvinchstr, mvwinchnstr, mvwinchstr — input an array of single-byte characters and renditions from a window

SYNOPSIS

```
EC #include <curses.h>

int mvinchstr(int y, int x, chtype *chstr, int n);
int mvwinchnstr(WINDOW *win, int y, int x, chtype *chstr, int n);
int mvwinchstr(WINDOW *win, int y, int x, chtype *chstr);
```

DESCRIPTION

Refer to *inchstr()*.

NAME

mvinnstr, mvinstr, mvwinnstr, mvwinstr — input a multi-byte character string from a window

SYNOPSIS

```
EC #include <curses.h>

int mvinnstr(int y, int x, char *str, int n);
int mvinstr(int y, int x, char *str);
int mvwinnstr(WINDOW *win, int y, int x, char *str, int n);
int mvwinstr(WINDOW *win, int y, int x, char *str);
```

DESCRIPTION

Refer to *innstr()*.

NAME

mvinnwstr, mvinwstr, mvwinnwstr, mvwinwstr — input a string of wide characters from a window

SYNOPSIS

```
EC #include <curses.h>

int mvinnwstr(int y, int x, wchar_t *wstr, int n);
int mvinwstr(int y, int x, wchar_t *wstr);
int mvwinnwstr(WINDOW *win, int y, int x, wchar_t *wstr, int n);
int mvwinwstr(WINDOW *win, int y, int x, wchar_t *wstr);
```

DESCRIPTION

Refer to *innwstr()*.

NAME

mvins_nwstr, mvins_wstr, mvwins_nwstr, mvwins_wstr — insert a wide-character string into a window

SYNOPSIS

```
EC #include <curses.h>

int mvins_nwstr(int y, int x, const wchar_t *wstr, int n);
int mvins_wstr(int y, int x, const wchar_t *wstr);
int mvwins_nwstr(WINDOW *win, int y, int x, const wchar_t *wstr,
                int n);
int mvwins_wstr(WINDOW *win, int y, int x, const wchar_t *wstr);
```

DESCRIPTION

Refer to [ins_nwstr\(\)](#).

NAME

mvins_wch, mvwins_wch — insert a complex character and rendition into a window

SYNOPSIS

```
EC #include <curses.h>
    int mvins_wch(int y, int x, const cchar_t *wch);
    int mvwins_wch(WINDOW *win, int y, int x, const cchar_t *wch);
```

DESCRIPTION

Refer to [ins_wch\(\)](#).

NAME

mvinsch, mvwinsch — insert a single-byte character and rendition into a window

SYNOPSIS

```
#include <curses.h>

int mvinsch(int y, int x, chtype ch);
int mvwinsch(WINDOW *win, int y, int x, chtype ch);
```

DESCRIPTION

Refer to *insch()*.

NAME

mvinsnstr, mvinsstr, mvwinsnstr, mvwinsstr — insert a multi-byte character string into a window

SYNOPSIS

```
EC #include <curses.h>

int mvinsnstr(int y, int x, const char *str, int n);
int mvinsstr(int y, int x, const char *str);
int mvwinsnstr(WINDOW *win, int y, int x, const char *str, int n);
int mvwinsstr(WINDOW *win, int y, int x, const char *str);
```

DESCRIPTION

Refer to *insnstr()*.

NAME

mvprintw, mvwprintw, printw, wprintw — print formatted output in window

SYNOPSIS

```
#include <curses.h>

int mvprintw(int y, int x, const char *fmt, ...);
int mvwprintw(WINDOW *win, int y, int x, const char *fmt, ...);
int printw(const char *fmt, ...);
int wprintw(WINDOW *win, const char *fmt, ...);
```

DESCRIPTION

These functions are analogous to *printf()*. The effect of these functions is as though *sprintf()* were used to format the string, and then *waddstr()* were used to add that multi-byte string to the current or specified window at the current or specified cursor position.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

addnstr(), *fprintf()* (in the XSH specification), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity and its name is changed from *printw()* to *mvprintw()*.

NAME

mvscanw, mvwscanw, scanw, wscanw — convert formatted input from a window

SYNOPSIS

```
#include <curses.h>

int mvscanw(int y, int x, const char *fmt, ...);
int mvwscanw(WINDOW *win, int y, int x, const char *fmt, ...);
int scanw(const char *fmt, ...);
int wscanw(WINDOW *win, const char *fmt, ...);
```

DESCRIPTION

These functions are similar to *scanf()*. Their effect is as though *mvwgetstr()* were called to get a multi-byte character string from the current or specified window at the current or specified cursor position, and then *sscanf()* were used to interpret and convert that string.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fscanf() (in the XSH specification), *getnstr()*, *mvprintw()*, *wcstombs()* (in the XSH specification), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity and its name is changed from *scanw()* to *mvscanw()*.

Issue 7

The prototypes for the *mvscanw()*, *mvwscanw()*, *scanw()*, and *wscanw()* functions are updated.

NAME

mvwin — move window

SYNOPSIS

```
#include <curses.h>

int mvwin(WINDOW *win, int y, int x);
```

DESCRIPTION

The *mvwin()* function moves the specified window so that its origin is at position (y, x) . If the move would cause any portion of the window to extend past any edge of the screen, the function fails and the window is not moved.

RETURN VALUE

Upon successful completion, the *mvwin()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The application should not move subwindows by calling *mvwin()*. Moving subwindows may cause processing in other subwindows in the parent window to become confused if the new location of the subwindow overlays or reveals part of another subwindow.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

derwin(), *doupdate()*, *is_linetouched()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

NAME

napms — suspend the calling process

SYNOPSIS

```
EC #include <curses.h>
   int napms(int ms);
```

DESCRIPTION

The *napms()* function takes at least *ms* milliseconds to return.

RETURN VALUE

The *napms()* function returns OK.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

A more reliable method of achieving a timed delay is the *nanosleep()* function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delay_output(), *nanosleep()* (in the XSH specification), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

newpad, pnoutrefresh, prefresh, subpad — pad management functions

SYNOPSIS

```
#include < curses.h>

WINDOW *newpad(int nlines, int ncols);
int pnoutrefresh(WINDOW *pad, int pminrow, int pmincol, int sminrow,
                int smincol, int smaxrow, int smaxcol);
int prefresh(WINDOW *pad, int pminrow, int pmincol, int sminrow,
            int smincol, int smaxrow, int smaxcol);
EC WINDOW *subpad(WINDOW *orig, int nlines, int ncols, int begin_y,
                int begin_x);
```

DESCRIPTION

The *newpad()* function creates a specialized window called a pad with *nlines* lines and *ncols* columns. A pad is like a window, except that it is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Automatic refreshes of pads (e.g., from scrolling or echoing of input) do not occur.

EC The *subpad()* function creates a specialized window within a pad (called the parent pad) called a subpad with *nlines* lines and *ncols* columns. Unlike *subwin()*, which uses screen coordinates, the subpad is created at position (*begin_y*, *begin_x*) within the parent pad. Changes made to either the parent or the subpad affect the other. The subpad must fit totally within the parent pad.

The *prefresh()* and *pnoutrefresh()* functions are analogous to *wrefresh()* and *wnoutrefresh()* except that they relate to pads instead of windows. The additional arguments indicate what part of the pad and screen are involved. The *pminrow* and *pmincol* arguments specify the origin of the rectangle to be displayed in the pad. The *sminrow*, *smincol*, *smaxrow*, and *smaxcol* arguments specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

RETURN VALUE

EC Upon successful completion, the *newpad()* and *subpad()* functions return a pointer to the pad data structure. Otherwise, they return a null pointer.

Upon successful completion, the *pnoutrefresh()* and *prefresh()* functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To refresh a pad, call *prefresh()* or *pnoutrefresh()*, not *wrefresh()*.

Although a subwindow and its parent pad may share memory representing characters in the pad, they need not share status information about what has changed in the pad. Therefore, after modifying a subwindow within a pad, it may be necessary to call *touchwin()* or *touchline()* on the pad before calling *prefresh()*.

Pads should be used whenever a window larger than the terminal screen is required.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

derwin(), *doupdate()*, *is_linetouched()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The *pnoutrefresh()* and *prefresh()* functions are merged with this entry.

The *subpad()* function is new in Issue 4.

Issue 7

Corrigendum U018/4 is applied, updating the DESCRIPTION of the *newpad()* and *subpad()* functions and adding use of pads to the APPLICATION USAGE section.

NAME

newterm — screen initialization function

SYNOPSIS

```
#include <curses.h>
```

```
SCREEN *newterm(const char *type, FILE *outfile, FILE *infile);
```

DESCRIPTION

Refer to *initscr()*.

NAME

newwin — create a new window

SYNOPSIS

```
#include <curses.h>
```

```
WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x);
```

DESCRIPTION

Refer to *derwin()*.

NAME

nl, nonl — enable/disable newline translation

SYNOPSIS

```
#include <curses.h>

int nl(void);
int nonl(void);
```

DESCRIPTION

The *nl()* function enables a mode in which <carriage-return> is translated to <newline> on input. The *nonl()* function disables the above translation. Initially, the above translation is enabled.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The default translation adapts the terminal to environments in which <newline> is the line termination character. However, by disabling the translation with *nonl()*, the application can sense the pressing of the <carriage-return> key.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *nl()* and *nonl()* functions is explicitly declared as **void**.

NAME

no — pointer page for functions with no prefix

DESCRIPTION

The no prefix indicates that a Curses function disables a mode. (The corresponding functions without the no prefix enable the same mode.)

The *no()** functions are discussed together with the corresponding functions that do not have these prefixes.

Note: The *nodelay()* function has an entry under its own name because there is no corresponding *delay()* function.

The *noqiflush()* and *notimeout()* functions have an entry under their own names because they precede the corresponding function without the no prefix in alphabetical order.

They are found in the following entries:

| Function | Refer to |
|-------------------|-----------------|
| <i>nocbreak()</i> | <i>cbreak()</i> |
| <i>noecho()</i> | <i>echo()</i> |
| <i>nonl()</i> | <i>nl()</i> |
| <i>noraw()</i> | <i>cbreak()</i> |

NAME

nocbreak, noraw — input mode control functions

SYNOPSIS

```
#include <curses.h>

int nocbreak(void);
int noraw(void);
```

DESCRIPTION

Refer to *cbreak()*.

NAME

nodelay — enable or disable block during read

SYNOPSIS

```
#include <curses.h>

int nodelay(WINDOW *win, bool bf);
```

DESCRIPTION

The *nodelay()* function specifies whether Delay Mode or No Delay Mode is in effect for the screen associated with the specified window. If *bf* is TRUE, this screen is set to No Delay Mode. If *bf* is FALSE, this screen is set to Delay Mode. The initial state is FALSE.

RETURN VALUE

Upon successful completion, the *nodelay()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), [getch\(\)](#), [halfdelay\(\)](#), [<curses.h>](#), XBD specification, Section 11.2, Parameters that Can be Set

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

NAME

noecho — enable/disable terminal echo

SYNOPSIS

```
#include <curses.h>

int noecho(void);
```

DESCRIPTION

Refer to *echo()*.

NAME

nonl — enable/disable newline translation

SYNOPSIS

```
#include <curses.h>
int nonl(void);
```

DESCRIPTION

Refer to *nl()*.

NAME

noqiflush, qiflush — enable/disable queue flushing

SYNOPSIS

```
EC #include <curses.h>
void noqiflush(void);
void qiflush(void);
```

DESCRIPTION

The *qiflush()* function causes all output in the display driver queue to be flushed whenever an interrupt key (interrupt, suspend, or quit) is pressed. The *noqiflush()* function causes no such flushing to occur. The default for the option is inherited from the display driver settings.

RETURN VALUE

These functions do not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Calling *qiflush()* provides faster response to interrupts, but causes Curses to have the wrong idea of what is on the screen. The same effect is achieved outside Curses using the NOFLSH local mode flag specified in the **XBD** specification (**General Terminal Interface**).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), *intrflush()*, [<curses.h>](#), **XBD** specification, Section 11.2, Parameters that Can be Set (NOFLSH flag)

CHANGE HISTORY

First released in Issue 4.

NAME

notimeout, timeout, wtimeout — control blocking on input

SYNOPSIS

```
EC #include <curses.h>

int notimeout(WINDOW *win, bool bf);
void timeout(int delay);
void wtimeout(WINDOW *win, int delay);
```

DESCRIPTION

The *notimeout()* function specifies whether Timeout Mode or No Timeout Mode is in effect for the screen associated with the specified window. If *bf* is TRUE, this screen is set to No Timeout Mode. If *bf* is FALSE, this screen is set to Timeout Mode. The initial state is FALSE.

The *timeout()* and *wtimeout()* functions set blocking or non-blocking read for the current or specified window based on the value of *delay*:

delay < 0 One or more blocking reads (indefinite waits for input) are used.

delay = 0 One or more non-blocking reads are used. Any Curses input function will fail if every character of the requested string is not immediately available.

delay > 0 Any Curses input function blocks for *delay* milliseconds and fails if there is still no input.

RETURN VALUE

Upon successful completion, the *notimeout()* function returns OK. Otherwise, it returns ERR.

The *timeout()* and *wtimeout()* functions do not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 3.5 (on page 22), *getch()*, *halfdelay()*, *nodelay()*, [<curses.h>](#), XBD specification, Section 11.2, Parameters that Can be Set

CHANGE HISTORY

First released in Issue 4.

NAME

overlay, overwrite — copy overlapped windows

SYNOPSIS

```
#include <curses.h>

int overlay(const WINDOW *srcwin, WINDOW *dstwin);
int overwrite(const WINDOW *srcwin, WINDOW *dstwin);
```

DESCRIPTION

These functions overlay *srcwin* on top of *dstwin*. The *srcwin* and *dstwin* arguments need not be the same size; only text where the two windows overlap is copied.

The *overwrite()* function copies characters as though a sequence of *win_wch()* and *wadd_wch()* were performed with the destination window's attributes and background attributes cleared.

The *overlay()* function does the same thing, except that whenever a character to be copied is the background character of the source window, *overlay()* does not copy the character but merely moves the destination cursor the width of the source background character.

If any portion of the overlaying window border is not the first column of a multi-column character, then all the column positions will be replaced with the background character and rendition before the overlay is done. If the default background character is a multi-column character when this occurs, then these functions fail.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[copywin\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The type of argument *srcwin()* is changed from **WINDOW *** to **WINDOW *CONST**.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

pair_content, PAIR_NUMBER — get information on a color pair

SYNOPSIS

```
EC    #include <curses.h>
      int pair_content(short pair, short *f, short *b);
      int PAIR_NUMBER(int value);
```

DESCRIPTION

Refer to [can_change_color\(\)](#).

NAME

pechochar, pecho_wchar — write a character and rendition and immediately refresh the pad

SYNOPSIS

```
EC #include <curses.h>

int pechochar(WINDOW *pad, chtype ch);
int pecho_wchar(WINDOW *pad, const cchar_t *wch);
```

DESCRIPTION

The *pechochar()* and *pecho_wchar()* functions output one character to a pad and immediately refresh the pad. They are equivalent to a call to *waddch()* or *wadd_wch()*, respectively, followed by a call to *prefresh()*. The last location of the pad on the screen is reused for the arguments to *prefresh()*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *pechochar()* function is only guaranteed to operate reliably on character sets in which each character fits into a single byte, whose attributes can be expressed using only constants with the A_ prefix.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

echochar(), *newpad()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

The second argument of *pechochar()* is changed to type **chtype** from **chtype***.

NAME

pnoutrefresh, prefresh — refresh pads

SYNOPSIS

```
#include <curses.h>

int pnoutrefresh(WINDOW *pad, int pminrow, int pmincol, int sminrow,
                int smincol, int smaxrow, int smaxcol);
int prefresh(WINDOW *pad, int pminrow, int pmincol, int sminrow,
             int smincol, int smaxrow, int smaxcol);
```

DESCRIPTION

Refer to *newpad()*.

NAME

printw — print formatted output in the current window

SYNOPSIS

```
#include <curses.h>

int printw(const char *fmt, ...);
```

DESCRIPTION

Refer to *mvprintw()*.

NAME

putp, tputs — output commands to the terminal

SYNOPSIS

```
EC      #include <term.h>
        int putp(const char *str);
        int tputs(const char *str, int affcnt, int (*putfunc)(int));
```

DESCRIPTION

These functions output commands contained in the **terminfo** database to the terminal.

The *putp()* function is equivalent to *tputs(str, 1, putchar)*. The output of *putp()* always goes to **stdout**, not to the *fildev* specified in *setupterm()*.

OB The *tputs()* function outputs *str* to the terminal. The *str* argument must be a **terminfo** string variable or the return value from *tiparm()* or *tparm()*. The *affcnt* argument is the number of lines affected, or 1 if not applicable. If the **terminfo** database indicates that the terminal in use requires padding after any command in the generated string, *tputs()* inserts pad characters into the string that is sent to the terminal, at positions indicated by the **terminfo** database. The *tputs()* function outputs each character of the generated string by calling the user-supplied function *putfunc* (see below).

The user-supplied function *putfunc* (specified as an argument to *tputs()*) is either *putchar()* or some other function with the same prototype. The *tputs()* function ignores the return value of *putfunc*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Changing the terminal attributes using these functions may cause the renditions of characters within a curses window to be altered on some terminals.

After use of any of these functions, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

doupdate(), *is_linetouched()*, *putchar()* (in the XSH specification), *tigetflag()*, **<term.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

putwin — dump window to a file

SYNOPSIS

```
EC #include <curses.h>
    int putwin(WINDOW *win, FILE *filep);
```

DESCRIPTION

Refer to *getwin()*.

NAME

qiflush — enable queue flushing

SYNOPSIS

```
EC #include <curses.h>
    void qiflush(void);
```

DESCRIPTION

Refer to *noqiflush()*.

NAME

raw — set Raw Mode

SYNOPSIS

```
#include <curses.h>

int raw(void);
```

DESCRIPTION

Refer to *cbreak()*.

NAME

redrawwin, wredrawln — line update status functions

SYNOPSIS

```
EC #include <curses.h>
int redrawwin(WINDOW *win);
int wredrawln(WINDOW *win, int beg_line, int num_lines);
```

DESCRIPTION

The *redrawwin()* and *wredrawln()* functions inform the implementation that some or all of the information physically displayed for the specified window may have been corrupted. The *redrawwin()* function marks the entire window; *wredrawln()* marks only *num_lines* lines starting at line number *beg_line*. The functions prevent the next refresh operation on that window from performing any optimization based on assumptions about what is physically displayed there.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *redrawwin()* and *wredrawln()* functions could be used in a text editor to implement a command that redraws some or all of the screen.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clearok(), *doupdate()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

refresh — refresh current window

SYNOPSIS

```
#include <curses.h>
int refresh(void);
```

DESCRIPTION

Refer to *doupdate()*.

NAME

reset_prog_mode, reset_shell_mode — restore program or shell terminal modes

SYNOPSIS

```
#include <curses.h>

int reset_prog_mode(void);
int reset_shell_mode(void);
```

DESCRIPTION

Refer to *def_prog_mode()*.

NAME

resetty, savetty — save/restore terminal mode

SYNOPSIS

```
#include <curses.h>

int resetty(void);
int savetty(void);
```

DESCRIPTION

The *resetty()* function restores the program mode as of the most recent call to *savetty()*.

The *savetty()* function saves the state that would be put in place by a call to *reset_prog_mode()*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

def_prog_mode(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The argument list for the *resetty()* and *savetty()* functions is explicitly declared as **void**.

NAME

restartterm — change terminal type

SYNOPSIS

```
EC #include <term.h>
int restartterm(char *term, int fildes, int *errret);
```

DESCRIPTION

Refer to [del_curterm\(\)](#).

NAME

ripoffline — reserve a line for a dedicated purpose

SYNOPSIS

```
EC #include <curses.h>
int ripoffline(int line, int (*init)(WINDOW *win, int columns));
```

DESCRIPTION

The *ripoffline()* function reserves a screen line for use by the application.

Any call to *ripoffline()* must precede the call to *initscr()* or *newterm()*. If *line* is positive, one line is removed from the beginning of *stdscr*; if *line* is negative, one line is removed from the end. Removal occurs during the subsequent call to *initscr()* or *newterm()*. When the subsequent call is made, the function pointed to by *init* is called with two arguments: a **WINDOW** pointer to the one-line window that has been allocated and an integer with the number of columns in the window. The initialization function cannot use the *LINES* and *COLS* external variables and cannot call *wrefresh()* or *doupdate()*, but may call *wnoutrefresh()*.

Up to five lines can be ripped off. Calls to *ripoffline()* above this limit have no effect but report success.

RETURN VALUE

The *ripoffline()* function returns OK.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Calling *slk_init()* reduces the size of the screen by one line if *initscr()* eventually uses a line from *stdscr* to emulate the soft labels. If *slk_init()* rips off a line, it thereby reduces by one the number of lines an application can reserve by subsequent calls to *ripoffline()*. Thus, portable applications that use soft label functions should not call *ripoffline()* more than four times.

When *initscr()* or *newterm()* calls the initialization function pointed to by *init*, the implementation may pass NULL for the **WINDOW** pointer argument *win*. This indicates inability to allocate a one-line window for the line that the call to *ripoffline()* ripped off. Portable applications should verify that *win* is not NULL before performing any operation on the window it represents.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

doupdate(), *initscr()*, *slk_attroff()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

savetty — save terminal mode

SYNOPSIS

```
#include <curses.h>
int savetty(void);
```

DESCRIPTION

Refer to *resetty()*.

NAME

scanw — convert formatted input from the current window

SYNOPSIS

```
#include <curses.h>

int scanw(const char *fmt, ...);
```

DESCRIPTION

Refer to *mvscanw()*.

NAME

scr_dump, scr_init, scr_restore, scr_set — screen file input/output functions

SYNOPSIS

```
EC #include <curses.h>

int scr_dump(const char *filename);
int scr_init(const char *filename);
int scr_restore(const char *filename);
int scr_set(const char *filename);
```

DESCRIPTION

The *scr_dump()* function writes the current contents of the virtual screen to the file named by *filename* in an unspecified format.

The *scr_restore()* function sets the virtual screen to the contents of the file named by *filename*, which must have been written using *scr_dump()*. The next refresh operation restores the screen to the way it looked in the dump file.

The *scr_init()* function reads the contents of the file named by *filename* and uses them to initialize the Curses data structures to what the terminal currently has on its screen. The next refresh operation bases any updates on this information, unless either of the following conditions is true:

- The terminal has been written to since the virtual screen was dumped to *filename*.
- The **terminfo** capabilities **rmcup** and **nrrmc** are defined for the current terminal.

The *scr_set()* function is a combination of *scr_restore()* and *scr_init()*. It tells the program that the information in the file named by *filename* is what is currently on the screen, and also what the program wants on the screen. This can be thought of as a screen inheritance function.

RETURN VALUE

On successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *scr_init()* function is called after *initscr()* or a *system()* call to share the screen with another process that has done a *scr_dump()* after its *endwin()* call.

To read a window from a file, call *getwin()*; to write a window to a file, call *putwin()*.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delscreen(), *doupdate()*, *endwin()*, *getwin()*, *open()* (in the XSH specification), *read()* (in the XSH specification), *write()* (in the XSH specification), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

sctrl, scroll, wscrl — scroll a Curses window

SYNOPSIS

```
#include <curses.h>
EC   int sctrl(int n);
      int scroll(WINDOW *win);
EC   int wscrl(WINDOW *win, int n);
```

DESCRIPTION

The *sctrl()* function scrolls *win* one line in the direction of the first line.

EC The *sctrl()* and *wscrl()* functions scroll the current or specified window. If *n* is positive, the window scrolls *n* lines toward the first line. Otherwise, the window scrolls *-n* lines toward the last line.

These functions do not change the cursor position. If scrolling is disabled for the current or specified window, these functions have no effect. The interaction of these functions with *setscrcg()* is currently unspecified.

RETURN VALUE

On successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

In a future version, the interaction of these functions with *setscrcg()* will be defined.

SEE ALSO

[<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

In previous versions, the *scroll()* function was described in an entry of its own. It has been merged with this entry in Issue 4. Its description has been rewritten for clarity, but otherwise its functionality is identical.

NAME

scrollok — enable or disable scrolling on a window

SYNOPSIS

```
#include <curses.h>

int scrollok(WINDOW *win, bool bf);
```

DESCRIPTION

Refer to *clearok()*.

NAME

set_curterm — set current terminal

SYNOPSIS

```
EC #include <term.h>
    TERMINAL *set_curterm(TERMINAL *nterm);
```

DESCRIPTION

Refer to [del_curterm\(\)](#).

NAME

set_term — switch between screens

SYNOPSIS

```
#include <curses.h>

SCREEN *set_term(SCREEN *new);
```

DESCRIPTION

The *set_term()* function switches between different screens. The *new* argument specifies the new current screen.

RETURN VALUE

Upon successful completion, the *set_term()* function returns a pointer to the previous screen. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This is the only function that manipulates **SCREEN** pointers; all other functions affect only the current screen.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.2](#) (on page 14), *initscr()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

NAME

setcchar — set **cchar_t** from a wide-character string and rendition

SYNOPSIS

```
EC #include <curses.h>

int setcchar(cchar_t *wcval, const wchar_t *wch, const attr_t attrs,
             short color_pair, const void *opts);
```

DESCRIPTION

The *setcchar()* function initializes the object pointed to by *wcval* according to the character attributes in *attrs*, the color pair in *color_pair*, and the wide-character string pointed to by *wch*.

The *opts* argument is reserved for definition in a future version. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, the *setcchar()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.3](#) (on page 15), *attroff()*, *can_change_color()*, *getcchar()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

setscrreg — define software scrolling region

SYNOPSIS

```
#include <curses.h>

int setscrreg(int top, int bot);
```

DESCRIPTION

Refer to *clearok()*.

NAME

setupterm — access the **terminfo** database

SYNOPSIS

```
EC #include <term.h>
int setupterm(char *term, int fildes, int *errret);
```

DESCRIPTION

Refer to [del_curterm\(\)](#).

NAME

slk_attroff, slk_attr_off, slk_attron, slk_attr_on, slk_attrset, slk_attr_set, slk_clear, slk_color, slk_init, slk_label, slk_noutrefresh, slk_refresh, slk_restore, slk_set, slk_touch, slk_wset — soft label functions

SYNOPSIS

```
EC #include <curses.h>

int slk_attroff(const chtype attrs);
int slk_attr_off(const attr_t attrs, void *opts);
int slk_attron(const chtype attrs);
int slk_attr_on(const attr_t attrs, void *opts);
int slk_attrset(const chtype attrs);
int slk_attr_set(const attr_t attrs, short color_pair_number,
                void *opts);
int slk_clear(void);
int slk_color(short color_pair_number);
int slk_init(int fmt);
char *slk_label(int labnum);
int slk_noutrefresh(void);
int slk_refresh(void);
int slk_restore(void);
int slk_set(int labnum, const char *label, int justify);
int slk_touch(void);
int slk_wset(int labnum, const wchar_t *label, int justify);
```

DESCRIPTION

The Curses interface manipulates the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, Curses takes over the bottom line of *stdscr*, reducing the size of *stdscr* and the value of the *LINES* external variable. There can be up to eight labels of up to eight display columns each.

To use soft labels, *slk_init()* must be called before *initscr()*, *newterm()*, or *ripoffline()* is called. If *initscr()* eventually uses a line from *stdscr* to emulate the soft labels, then *fmt* determines how the labels are arranged on the screen. Setting *fmt* to 0 indicates a 3-2-3 arrangement of the labels; 1 indicates a 4-4 arrangement. Other values for *fmt* are unspecified.

The *slk_init()* function has the effect of calling *ripoffline()* to reserve one screen line to accommodate the requested format.

The *slk_set()* and *slk_wset()* functions specify the text of soft label number *labnum*, within the range from 1 to and including 8. The *label* argument is the string to be put on the label. With *slk_set()* and *slk_wset()*, the width of the label is limited to eight column positions. A null string or a null pointer specifies a blank label. The *justify* argument can have the following values to indicate how to justify *label* within the space reserved for it:

- 0 Align the start of *label* with the start of the space.
- 1 Center *label* within the space.
- 2 Align the end of *label* with the end of the space.

The *slk_refresh()* and *slk_noutrefresh()* functions correspond to the *wrefresh()* and *wnoutrefresh()* functions.

The *slk_label()* function obtains soft label number *labnum*.

The *slk_clear()* function immediately clears the soft labels from the screen.

The *slk_restore()* function immediately restores the soft labels to the screen after a call to *slk_clear()*.

The *slk_touch()* function forces all the soft labels to be output the next time *slk_noutrefresh()* or *slk_refresh()* is called.

The *slk_attron()*, *slk_attrset()*, and *slk_attroff()* functions correspond to *attron()*, *attrset()*, and *attroff()*. They have an effect only if soft labels are simulated on the bottom line of the screen.

The *slk_attr_off()*, *slk_attr_on()*, *slk_attr_set()*, and *slk_color()* functions correspond to *slk_attroff()*, *slk_attrond()*, *slk_attrset()*, and *color_set()* and thus support the attribute constants with the *WA_* prefix and *color*.

The *opts* argument is reserved for definition in a future version. Currently, the application must provide a null pointer as *opts*.

RETURN VALUE

Upon successful completion, the *slk_label()* function returns the requested label with leading and trailing <blank>s stripped. Otherwise, it returns a null pointer.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

When using multi-byte character sets, applications should check the width of the string by calling *mbstowcs()* and then *wcswidth()* before calling *slk_set()*. When using wide characters, applications should check the width of the string by calling *wcswidth()* before calling *slk_set()*.

Since the number of columns that a wide-character string will occupy is codeset-specific, call *wcwidth()* and *wcswidth()* to check the number of column positions in the string before calling *slk_wset()*.

Most applications would use *slk_noutrefresh()* because a *wrefresh()* is likely to follow soon.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

attr_get(), *attroff()*, *delscreen()*, *mbstowcs()* (in the XSH specification), *ripoffline()*, *wcswidth()* (in the XSH specification), <[curses.h](#)>

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

This entry is rewritten to include the color handling functions.

NAME

standend, standout, wstandend, wstandout — set and clear window attributes

SYNOPSIS

```
#include <curses.h>

int standend(void);
int standout(void);
int wstandend(WINDOW *win);
int wstandout(WINDOW *win);
```

DESCRIPTION

The *standend()* and *wstandend()* functions turn off all attributes of the current or specified window.

The *standout()* and *wstandout()* functions turn on the standout attribute of the current or specified window.

RETURN VALUE

These functions always return 1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[attroff\(\)](#), [attr_get\(\)](#), [<curses.h>](#)

CHANGE HISTORY

Derived from the *attroff()* entry in Issue 3. The entry is reworded for clarity, but otherwise the functionality is identical to previous version.

NAME

start_color — initialize use of colors on terminal

SYNOPSIS

```
EC #include <curses.h>
   int start_color(void);
```

DESCRIPTION

Refer to [can_change_color\(\)](#).

NAME

stdscr — default window

SYNOPSIS

```
EC #include <curses.h>
extern WINDOW *stdscr;
```

DESCRIPTION

The external variable *stdscr* specifies the default window used by functions that do not specify a window using an argument of type **WINDOW ***. Other windows may be created using *newwin()*.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

derwin(), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

subpad — create a subwindow in a pad

SYNOPSIS

```
EC #include <curses.h>
WINDOW *subpad(WINDOW *orig, int nlines, int ncols, int begin_y,
               int begin_x);
```

DESCRIPTION

Refer to *newpad()*.

NAME

subwin — create a subwindow

SYNOPSIS

```
#include <curses.h>
```

```
WINDOW *subwin(WINDOW *orig, int nlines, int ncols, int begin_y,  
               int begin_x);
```

DESCRIPTION

Refer to *derwin()*.

NAME

syncok, wcursyncup, wsyncdown, wsyncup — synchronize a window with its parents or children

SYNOPSIS

```
EC #include <curses.h>

int syncok(WINDOW *win, bool bf);
void wcursyncup(WINDOW *win);
void wsyncdown(WINDOW *win);
void wsyncup(WINDOW *win);
```

DESCRIPTION

The *syncok()* function determines whether all ancestors of the specified window are implicitly touched whenever there is a change in the window. If *bf* is TRUE, such implicit touching occurs. If *bf* is FALSE, such implicit touching does not occur. The initial state is FALSE.

The *wcursyncup()* function updates the current cursor position of the ancestors of *win* to reflect the current cursor position of *win*.

The *wsyncdown()* function touches *win* if any ancestor window has been touched.

The *wsyncup()* function unconditionally touches all ancestors of *win*.

RETURN VALUE

Upon successful completion, the *syncok()* function returns OK. Otherwise, it returns ERR.

The other functions do not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications seldom call *wsyncdown()* because it is called by all refresh operations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[doupdate\(\)](#), [is_linetouched\(\)](#), [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

NAME

termattrs — get supported terminal video attributes

SYNOPSIS

```
EC #include <curses.h>
    chtype termattrs(void);
    attr_t term_attrs(void);
```

DESCRIPTION

The *termattrs()* function extracts the video attributes of the current terminal which is supported by the **chtype** data type.

The *term_attrs()* function extracts information for the video attributes of the current terminal which is supported for a **cchar_t**.

RETURN VALUE

The *termattrs()* function returns a logical OR of A_ values of all video attributes supported by the terminal.

The *term_attrs()* function returns a logical OR of WA_ values of all video attributes supported by the terminal.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

attroff(), *attr_get()*, **<curses.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS; rewritten for clarity.

NAME

termname — get terminal name

SYNOPSIS

```
EC #include <curses.h>
char *termname(void);
```

DESCRIPTION

The *termname()* function obtains the terminal name as recorded by *setupterm()*.

RETURN VALUE

The *termname()* function returns a pointer to the terminal name.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 7.1.1](#) (on page 338), *del_curterm()*, *getenv()* (in the **XSH** specification), *initscr()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

tigetflag, tigetnum, tigetstr, tiparm, tparm — retrieve or process capabilities from the **terminfo** database

SYNOPSIS

```
EC #include <term.h>

int tigetflag(const char *capname);
int tigetnum(const char *capname);
char *tigetstr(const char *capname);
char *tiparm(const char *cap, ...);
EC OB char *tparm(const char *cap, long p1, long p2, long p3, long p4,
               long p5, long p6, long p7, long p8, long p9);
```

DESCRIPTION

The *tigetflag()*, *tigetnum()*, and *tigetstr()* functions obtain boolean, numeric, and string capabilities, respectively, from the selected record of the **terminfo** database. For each capability, the value to use as *capname* appears in the **Capname** column in the table in [Section 7.1.3](#) (on page 340).

OB The *tiparm()* and *tparm()* functions take as *cap* a string capability. If *cap* is parameterized (as described in [Section A.1.2](#), on page 354), these functions resolve the parameterization as described below.

If the parameterized string refers to one or more of the parameters %p1 through %p9, then *tiparm()* fetches one argument for each %pN parameter, in order of *N* (that is, the first argument after *cap* is fetched for %p1, the second for %p2, and so on), and uses the corresponding argument value when pushing the %pN parameter on to the stack. The results are undefined if there are insufficient arguments for the parameterized string. If a %pN parameter is used in a string context (for example, if it is popped using %l or %s), the corresponding argument is fetched as type **char ***; otherwise, the argument is fetched as type **int**. If a %pN parameter is used more than once, at least one of the uses is in a string context, and the uses are not all in a string context, then the behavior is undefined. If parameter %pN is used and any of the lower numbered parameters, from %p1 to %p(N-1), are not used, the arguments corresponding to the unused parameters are fetched as type **int**.

OB If the parameterized string refers to one or more of the parameters %p1 through %p9, then *tparm()* uses the values of p1 through p9, respectively, when pushing the parameter on to the stack. If any of the parameters %p1 through %p9 is used in a string context (for example, if it is popped using %l or %s), the behavior is implementation-defined.

RETURN VALUE

Upon successful completion, the *tigetflag()*, *tigetnum()*, and *tigetstr()* functions return the specified capability. The *tigetflag()* function returns -1 if *capname* is not a boolean capability. The *tigetnum()* function returns -2 if *capname* is not a numeric capability. The *tigetstr()* function returns (**char ***)-1 if *capname* is not a string capability.

OB Upon successful completion, the *tiparm()* and *tparm()* functions return the capability pointed to by *cap* with parameterization resolved. Otherwise, they return a null pointer.

OB The return value from *tiparm()* and *tparm()* may point to static data which may be overwritten by a subsequent call to either function.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For parameterized string capabilities, the application should pass the return value from *tigetstr()* to *tiparm()*, as described above. The *tiparm()* function is a replacement for the obsolescent *tparm()* function which cannot support string parameters on implementations where converting **char *** pointers to **long int** and back does not preserve their values.

Note that when converting old code to use *tiparm()* instead of *tparm()*, it is important to ensure that numeric argument values are passed to *tiparm()* as type **int**, or a type that promotes to **int**. With *tparm()*, numeric arguments could have any integer type and they would be converted to the correct type (**long int**) courtesy of the function prototype. However, *tiparm()* has a prototype that ends with an ellipsis, and therefore no such conversion is performed.

Applications intending to send terminal capabilities directly to the terminal (which should only be done using *tputs()* or *putp()*) instead of using Curses, normally should obey the following rules:

- Call *reset_shell_mode()* to restore the display modes before exiting.
- If using cursor addressing, output **enter_ca_mode** upon startup and output **exit_ca_mode** before exiting.
- If using shell escapes, output **exit_ca_mode** and call *reset_shell_mode()* before calling the shell; call *reset_prog_mode()* and output **enter_ca_mode** after returning from the shell.

All parameterized terminal capabilities defined in this document can be passed to *tiparm()*. All parameterized terminal capabilities defined in this document except **pkey_key**, **pkey_local**, **pkey_plab**, **pkey_xmit**, and **plab_norm** can be passed to *tparm()*. Some implementations (those where **char *** can be converted to **long int** and back without loss) might also allow **pkey_key**, **pkey_local**, **pkey_plab**, **pkey_xmit**, and **plab_norm** to be passed to *tparm()*.

Some implementations create their own capabilities, create capabilities for non-terminal devices, and redefine the capabilities in this document. These practices are non-conforming because it may be that *tiparm()* and *tparm()* cannot parse these user-defined strings.

Applications should use the *tigetflag()*, *tigetnum()*, *tigetstr()*, and *tiparm()* functions instead of the withdrawn *tgetent()*, *tgetflag()*, *tgetnum()*, *tgetstr()*, and *tgoto()* functions. Note that these replacement functions are only required to be supported on implementations supporting X/Open Enhanced Curses.

RATIONALE

The *tiparm()* function does not require that if parameter **%pN** is used in the parameterized string, **%p1** through **%p(N-1)** must also be used. This is because some capabilities may have no use for some arguments in the definition for a specific terminal. An example is given in [Section A.1.7](#) (on page 358) for **sgr** where the terminal has *altcharset* but does not have protect mode, and so the parameterized string would use **%p9** but would not need to use **%p8**.

The arguments corresponding to unused parameters are fetched as type **int**, because numeric parameters are far more common than string parameters. If the need should arise for a string parameter to be (effectively) unused for a specific terminal, this can be handled by making the parameterized string push the parameter, pop it with **%1**, and then not use the length that was pushed by **%1**. This is sufficient for *tiparm()* to see the parameter being used in a string context, so that it will still expect the corresponding argument to have type **char ***.

FUTURE DIRECTIONS

None.

SEE ALSO

def_prog_mode(), *putp()*, **<term.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 7

The prototypes for the *tigetflag()*, *tigetnum()*, and *tigetstr()* functions are updated.

NAME

timeout — control blocking on input

SYNOPSIS

```
EC #include <curses.h>
    void timeout(int delay);
```

DESCRIPTION

Refer to *notimeout()*.

NAME

tiparm — format **terminfo** string capability

SYNOPSIS

```
EC #include <term.h>
char *tiparm(const char *cap, ...);
```

DESCRIPTION

Refer to *tigetflag()*.

NAME

touchline, touchwin — window refresh control functions

SYNOPSIS

```
#include <curses.h>
```

```
EC int touchline(WINDOW *win, int start, int count);  
   int touchwin(WINDOW *win);
```

DESCRIPTION

Refer to [is_linetouched\(\)](#).

NAME

tparm — format **terminfo** string capability

SYNOPSIS

```
EC OB #include <term.h>
char *tparm(const char *cap, long p1, long p2, long p3, long p4,
            long p5, long p6, long p7, long p8, long p9);
```

DESCRIPTION

Refer to *tigetflag()*.

NAME

tputs — output commands to the terminal

SYNOPSIS

```
EC #include <curses.h>
    int tputs(const char *str, int affcnt, int (*putfunc)(int));
```

DESCRIPTION

Refer to *putp()*.

NAME

typeahead — control checking for typeahead

SYNOPSIS

```
#include <curses.h>

int typeahead(int fildes);
```

DESCRIPTION

The *typeahead()* function controls the detection of typeahead during a refresh, based on the value of *fildes*:

- If *fildes* is a valid file descriptor, typeahead is enabled during refresh; Curses periodically checks *fildes* for input and aborts the refresh if any character is available. (This is the initial setting, and the typeahead file descriptor corresponds to the input file associated with the screen created by *initscr()* or *newterm()*.) The value of *fildes* need not be the file descriptor on which the refresh is occurring.
- If *fildes* is `-1`, Curses does not check for typeahead during refresh.

RETURN VALUE

Upon successful completion, the *typeahead()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 3.5 (on page 22), *doupdate()*, *getch()*, *initscr()*, `<curses.h>`, XBD specification, Section 11.2, Parameters that Can be Set

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The RETURN VALUE section now states that the function returns OK on success and ERR on failure. No return values were defined in previous versions.

NAME

unctrl — generate printable representation of a character

SYNOPSIS

```
#include <unctrl.h>

char *unctrl(chtype c);
```

DESCRIPTION

The *unctrl()* function generates a character string that is a printable representation of *c*. If *c* is a control character, it is converted to the ^X notation. If *c* contains rendition information, the effect is undefined.

RETURN VALUE

Upon successful completion, the *unctrl()* function returns the generated string. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

keyname(), *wunctrl()*, [<unctrl.h>](#)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is rewritten for clarity.

The RETURN VALUE section now states that the function may return a null pointer. This condition was not specified in previous versions.

NAME

ungetch, unget_wch — push a character onto the input queue

SYNOPSIS

```
EC #include <curses.h>
   int ungetch(int ch);
   int unget_wch(const wchar_t wch);
```

DESCRIPTION

The *ungetch()* function pushes the single-byte character *ch* onto the head of the input queue.

The *unget_wch()* function pushes the wide character *wch* onto the head of the input queue.

One character of push-back is guaranteed. The result of successive calls without an intervening call to *getch()* or *get_wch()* are unspecified.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Section 3.5](#) (on page 22), *getch()*, *get_wch()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

NAME

untouchwin — window refresh control function

SYNOPSIS

```
EC #include <curses.h>
    int untouched(WINDOW *win);
```

DESCRIPTION

Refer to [is_linetouched\(\)](#).

NAME

use_env — specify source of screen size information

SYNOPSIS

```
EC #include <curses.h>
void use_env(bool boolval);
```

DESCRIPTION

The *use_env()* function specifies the technique by which the implementation determines the size of the screen. If *boolval* is FALSE, the implementation uses the values of *lines* and *columns* specified in the **terminfo** database. If *boolval* is TRUE, the implementation uses the *LINES* and *COLUMNS* environment variables. The initial value is TRUE.

Any call to *use_env()* must precede calls to *initscr()*, *newterm()*, or *setupterm()*.

RETURN VALUE

The function does not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

del_curterm(), *initscr()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

The first argument is changed from **char** *bool* to **bool** *boolval*.

NAME

vidattr, vid_attr, vidputs, vid_puts — output attributes to the terminal

SYNOPSIS

```
EC #include <curses.h>

int vidattr(chtype attr);
int vid_attr(attr_t attr, short color_pair_number, void *opt);
int vidputs(chtype attr, int (*putfunc)(int));
int vid_puts(attr_t attr, short color_pair_number, void *opt, int
  (*putfunc)(int));
```

DESCRIPTION

These functions output commands to the terminal that change the terminal's attributes.

If the **terminfo** database indicates that the terminal in use can display characters in the rendition specified by *attr*, then *vidattr()* outputs one or more commands to request that the terminal display subsequent characters in that rendition. The function outputs by calling *putchar()*. The *vidattr()* function neither relies on nor updates the model which Curses maintains of the prior rendition mode.

The *vidputs()* function computes the same terminal output string that *vidattr()* does, based on *attr*, but *vidputs()* outputs by calling the user-supplied function *putfunc*. The *vid_attr()* and *vid_puts()* functions correspond to *vidattr()* and *vidputs()* respectively, but take a set of arguments, one of type **attr_t** for the attributes, short for the color pair number and a **void ***, and thus support the attribute constants with the *WA_* prefix.

The *opts* argument is reserved for definition in a future version. Currently, the application must provide a null pointer as *opts*.

The user-supplied function *putfunc* (which can be specified as an argument to either *vidputs()* or *vid_puts()*) is either *putchar()* or some other function with the same prototype. Both the *vidputs()* and the *vid_puts()* function ignore the return value of *putfunc*.

RETURN VALUE

Upon successful completion, these functions return OK. Otherwise, they return ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

After use of any of these functions, the model Curses maintains of the state of the terminal might not match the actual state of the terminal. The application should touch and refresh the window before resuming conventional use of Curses.

Use of these functions requires that the application contain so much information about a particular class of terminal that it defeats the purpose of using Curses.

On some terminals, a command to change rendition conceptually occupies space in the screen buffer (with or without width). Thus, a command to set the terminal to a new rendition would change the rendition of some characters already displayed.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

douupdate(), *is_linetouched()*, *putchar()* (in the XSH specification), *putwchar()* (in the XSH specification), *tigetflag()*, **<curses.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

This entry is rewritten to include the color handling functions.

NAME

vline — draw vertical line

SYNOPSIS

```
EC #include <curses.h>
   int vline(chtype ch, int n);
```

DESCRIPTION

Refer to [hline\(\)](#).

NAME

vline_set — draw vertical line from complex character and rendition

SYNOPSIS

```
EC #include <curses.h>
   int vline_set(const cchar_t *ch, int n);
```

DESCRIPTION

Refer to [hline_set\(\)](#).

NAME

vw_printw — print formatted output in window

SYNOPSIS

```
EC #include <stdarg.h>
   #include <curses.h>

   int vw_printw(WINDOW *, const char *, va_list varglst);
```

DESCRIPTION

The *vw_printw()* function achieves the same effect as *wprintw()* using a variable argument list. The third argument is a **va_list**, as defined in **<stdarg.h>**.

RETURN VALUE

Upon successful completion, the *vw_printw()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications should use the *vw_printw()* function instead of the withdrawn *wprintw()* function. Note that this replacement function is only required to be supported on implementations supporting X/Open Enhanced Curses.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mvprintw(), *fprintf()* (in the XSH specification), **<curses.h>**, **<stdarg.h>** (in the XBD specification)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS.

Issue 7

The prototype for the *vw_printw()* function is updated.

NAME

vw_scanw — convert formatted input from a window

SYNOPSIS

```
EC #include <stdarg.h>
   #include <curses.h>

   int vw_scanw(WINDOW *, const char *, va_list varglist);
```

DESCRIPTION

The *vw_scanw()* function achieves the same effect as *wscanw()* using a variable argument list. The third argument is a **va_list**, as defined in **<stdarg.h>**.

RETURN VALUE

Upon successful completion, the *vw_scanw()* function returns OK. Otherwise, it returns ERR.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications should use the *vw_scanw()* function instead of the withdrawn *wscanw()* function. Note that this replacement function is only required to be supported on implementations supporting X/Open Enhanced Curses.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fscanf() (in the XSH specification), *mvscanw()*, **<curses.h>**, **<stdarg.h>** (in the XBD specification)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

Corrections made to the SYNOPSIS and APPLICATION USAGE.

Issue 7

The prototype for the *vw_scanw()* function is updated.

NAME

w — pointer page for functions with w prefix

DESCRIPTION

Most uses of the w prefix indicate that a Curses function takes a *win* argument that specifies the affected window.

Note: The *wunctrl()* function is an exception to this rule and has an entry under its own name.

(The corresponding functions without the w prefix operate on the current window.)

The w functions are discussed together with the corresponding functions without the w prefix. They are as follows.

Note: The asterisk (*) denotes that there is no corresponding function without the w prefix.

| Function | Refer to |
|-----------------------|----------------------|
| <i>waddch()</i> | <i>addch()</i> |
| <i>waddchnstr()</i> | <i>addchnstr()</i> |
| <i>waddchstr()</i> | <i>addchstr()</i> |
| <i>waddnstr()</i> | <i>addnstr()</i> |
| <i>waddstr()</i> | <i>addstr()</i> |
| <i>waddnwstr()</i> | <i>addnwstr()</i> |
| <i>waddwstr()</i> | <i>addwstr()</i> |
| <i>wadd_wch()</i> | <i>add_wch()</i> |
| <i>wadd_wchnstr()</i> | <i>add_wchnstr()</i> |
| <i>wadd_wchstr()</i> | <i>add_wchstr()</i> |
| <i>wattroff()</i> | <i>attroff()</i> |
| <i>wattron()</i> | <i>attroff()</i> |
| <i>wattrset()</i> | <i>attroff()</i> |
| <i>wattr_get()</i> | <i>attr_get()</i> |
| <i>wattr_off()</i> | <i>attr_get()</i> |
| <i>wattr_on()</i> | <i>attr_get()</i> |
| <i>wattr_set()</i> | <i>attr_get()</i> |
| <i>wbkgd()</i> | <i>bkgd()</i> |
| <i>wbkgdset()</i> | <i>bkgd()</i> |
| <i>wbkgdset()</i> | <i>bkgdset()</i> |
| <i>wborder()</i> | <i>border()</i> |
| <i>wborder_set()</i> | <i>border_set()</i> |
| <i>wchgat()</i> | <i>chgat()</i> |
| <i>wclear()</i> | <i>clear()</i> |
| <i>wclrtoobot()</i> | <i>clrtoobot()</i> |
| <i>wclrtoeol()</i> | <i>clrtoeol()</i> |
| <i>wcursyncup()</i> * | <i>syncok()</i> |
| <i>wdelch()</i> | <i>delch()</i> |
| <i>wdeleteln()</i> | <i>deleteln()</i> |
| <i>wechochar()</i> | <i>echochar()</i> |
| <i>wecho_wchar()</i> | <i>echo_wchar()</i> |
| <i>werase()</i> | <i>clear()</i> |
| <i>wgetbkgrnd()</i> | <i>bkgrnd()</i> |
| <i>wgetch()</i> | <i>getch()</i> |
| <i>wgetnstr()</i> | <i>getnstr()</i> |
| <i>wgetn_wstr()</i> | <i>getn_wstr()</i> |
| <i>wgetstr()</i> | <i>getnstr()</i> |

| Function | Refer to |
|------------------------|-------------------------|
| <i>wget_wch()</i> | <i>get_wch()</i> |
| <i>wget_wstr()</i> | <i>getn_wstr()</i> |
| <i>whline()</i> | <i>hline()</i> |
| <i>whline_set()</i> | <i>hline_set()</i> |
| <i>winch()</i> | <i>inch()</i> |
| <i>winchnstr()</i> | <i>inchnstr()</i> |
| <i>winchstr()</i> | <i>inchnstr()</i> |
| <i>winnstr()</i> | <i>innstr()</i> |
| <i>winnwstr()</i> | <i>innwstr()</i> |
| <i>winsch()</i> | <i>insch()</i> |
| <i>winsdelln()</i> | <i>insdelln()</i> |
| <i>winsertln()</i> | <i>insertln()</i> |
| <i>winsnstr()</i> | <i>insnstr()</i> |
| <i>winsstr()</i> | <i>insnstr()</i> |
| <i>winstr()</i> | <i>innstr()</i> |
| <i>wins_nwstr()</i> | <i>ins_nwstr()</i> |
| <i>wins_wch()</i> | <i>ins_wch()</i> |
| <i>wins_wstr()</i> | <i>ins_nwstr()</i> |
| <i>winwstr()</i> | <i>innwstr()</i> |
| <i>win_wch()</i> | <i>in_wch()</i> |
| <i>win_wchnstr()</i> | <i>in_wchnstr()</i> |
| <i>win_wchstr()</i> | <i>in_wchnstr()</i> |
| <i>wmove()</i> | <i>move()</i> |
| <i>wnoutrefresh()*</i> | <i>doupdate()</i> |
| <i>wprintw()</i> | <i>moprintw()</i> |
| <i>wredrawln()</i> | <i>redrawwin()</i> |
| <i>wrefresh()</i> | <i>doupdate()</i> |
| <i>wscanw()</i> | <i>mvscanw()</i> |
| <i>wscrl()</i> | <i>scr1()</i> |
| <i>wsetscrreg()</i> | <i>clearok()</i> |
| <i>wstandend()</i> | <i>standend()</i> |
| <i>wstandout()</i> | <i>standend()</i> |
| <i>wsyncdown()*</i> | <i>syncok()</i> |
| <i>wsyncup()*</i> | <i>syncok()</i> |
| <i>wtimeout()</i> | <i>notimeout()</i> |
| <i>wtouchln()*</i> | <i>is_linetouched()</i> |
| <i>wvline()</i> | <i>hline()</i> |
| <i>wvline_set()</i> | <i>hline_set()</i> |

NAME

wadd_wch — add a complex character and rendition to a window

SYNOPSIS

```
EC #include <curses.h>
    int wadd_wch(WINDOW *win, const cchar_t *wch);
```

DESCRIPTION

Refer to [add_wch\(\)](#).

NAME

wadd_wchnstr, wadd_wchstr — add an array of complex characters and renditions to a window

SYNOPSIS

```
EC #include <curses.h>
    int wadd_wchnstr(WINDOW *win, const cchar_t *wchstr, int n);
    int wadd_wchstr(WINDOW *win, const cchar_t *wchstr);
```

DESCRIPTION

Refer to [add_wchnstr\(\)](#).

NAME

waddch — add a single-byte character and rendition to a window and advance the cursor

SYNOPSIS

```
#include <curses.h>

int waddch(WINDOW *win, const chtype ch);
```

DESCRIPTION

Refer to [addch\(\)](#).

NAME

waddchstr, waddchnstr — add string of single-byte characters and renditions to a window

SYNOPSIS

```
#include <curses.h>

int waddchstr(WINDOW *win, const chtype *chstr);
EC int waddchnstr(WINDOW *win, const chtype *chstr, int n);
```

DESCRIPTION

Refer to *addchstr()*.

NAME

waddnstr, waddstr — add a string of multi-byte characters without rendition to a window and advance cursor

SYNOPSIS

```
EC #include <curses.h>
    int waddnstr(WINDOW *win, const char *str, int n);
    int waddstr(WINDOW *win, const char *str);
```

DESCRIPTION

Refer to [addnstr\(\)](#).

NAME

waddnwstr, waddwstr — add a wide-character string to a window and advance the cursor

SYNOPSIS

```
EC #include <curses.h>
int waddnwstr(WINDOW *win, const wchar_t *wstr, int n);
int waddwstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

Refer to *addnwstr()*.

NAME

wattr_get, wattr_off, wattr_on, wattr_set — window attribute control functions

SYNOPSIS

```
EC #include <curses.h>

int wattr_get(WINDOW *win, attr_t *attrs, short *color_pair_number,
              void *opts);
int wattr_off(WINDOW *win, attr_t attrs, void *opts);
int wattr_on(WINDOW *win, attr_t attrs, void *opts);
int wattr_set(WINDOW *win, attr_t attrs, short color_pair_number,
              void *opts);
```

DESCRIPTION

Refer to [attr_get\(\)](#).

NAME

wattroff, wattron, wattrset — restricted window attribute control functions

SYNOPSIS

```
#include <curses.h>

int wattroff(WINDOW *win, int attrs);
int wattron(WINDOW *win, int attrs);
int wattrset(WINDOW *win, int attrs);
```

DESCRIPTION

Refer to *wattroff()*.

NAME

`wbkgd`, `wbkgdset` — turn off the previous background attributes, logical OR the requested attributes into the window rendition, and set or get background character and rendition using a single-byte character

SYNOPSIS

```
EC #include <curses.h>
    int wbkgd(WINDOW *win, chtype ch);
    void wbkgdset(WINDOW *win, chtype ch);
```

DESCRIPTION

Refer to *[bkgd\(\)](#)*.

NAME

wbkgrnd, wbkgrndset, wgetbkgrnd — turn off the previous background attributes, OR the requested attributes into the window rendition, and set or get background character and rendition using a complex character

SYNOPSIS

```
EC #include <curses.h>
    int wbkgrnd(WINDOW *win, const cchar_t *wch);
    void wbkgrndset(WINDOW *win, const cchar_t *wch);
    int wgetbkgrnd(WINDOW *win, cchar_t *wch);
```

DESCRIPTION

Refer to *bkgrnd()*.

NAME

wborder — draw borders from single-byte characters and renditions

SYNOPSIS

```
EC #include <curses.h>
    int wborder(WINDOW *win, chtype ls, chtype rs, chtype ts, chtype bs,
               chtype tl, chtype tr, chtype bl, chtype br);
```

DESCRIPTION

Refer to *border()*.

NAME

wborder_set — draw borders from complex characters and renditions

SYNOPSIS

```
EC #include <curses.h>

int wborder_set(WINDOW *win, const cchar_t *ls, const cchar_t *rs,
               const cchar_t *ts, const cchar_t *bs,
               const cchar_t *tl, const cchar_t *tr,
               const cchar_t *bl, const cchar_t *br);
```

DESCRIPTION

Refer to [border_set\(\)](#).

NAME

wchgat — change renditions of characters in a window

SYNOPSIS

```
EC #include <curses.h>
    int chgat(int n, attr_t attr, short color, const void *opts);
```

DESCRIPTION

Refer to *chgat()*.

NAME

wclear, werase — clear a window

SYNOPSIS

```
#include <curses.h>

int wclear(WINDOW *win);
int werase(WINDOW *win);
```

DESCRIPTION

Refer to *clear()*.

NAME

wclrrobot — clear from cursor to end of window

SYNOPSIS

```
#include <curses.h>

int wclrrobot(WINDOW *win);
```

DESCRIPTION

Refer to *clrrobot()*.

NAME

wclrtoeol — clear from cursor to end of line

SYNOPSIS

```
#include <curses.h>

int wclrtoeol(WINDOW *win);
```

DESCRIPTION

Refer to *clrtoeol()*.

NAME

wcolor_set — window attribute control functions

SYNOPSIS

```
EC #include <curses.h>
    int wcolor_set(WINDOW *win, short color_pair_number, void *opts);
```

DESCRIPTION

Refer to [attr_get\(\)](#).

NAME

wcursyncup — synchronize a window with its parents or children

SYNOPSIS

```
EC #include <curses.h>
    void wcursyncup(WINDOW *win);
```

DESCRIPTION

Refer to *syncok()*.

NAME

wdelch — delete a character from a window

SYNOPSIS

```
#include <curses.h>

int wdelch(WINDOW *win);
```

DESCRIPTION

Refer to *delch()*.

NAME

wdeleteln — delete lines in a window

SYNOPSIS

```
#include <curses.h>

int wdeleteln(WINDOW *win);
```

DESCRIPTION

Refer to *deleteln()*.

NAME

wecho_wchar — write a complex character and immediately refresh the window

SYNOPSIS

```
EC #include <curses.h>
    int wecho_wchar(WINDOW *win, const cchar_t *wch);
```

DESCRIPTION

Refer to [echo_wchar\(\)](#).

NAME

wechochar — echo single-byte character and rendition to a window and refresh

SYNOPSIS

```
EC #include <curses.h>
   int wechochar(WINDOW *win, const chtype ch);
```

DESCRIPTION

Refer to *echochar()*.

NAME

wget_wch — get a wide character from a terminal

SYNOPSIS

```
EC #include <curses.h>
   int wget_wch(WINDOW *win, wint_t *ch);
```

DESCRIPTION

Refer to [get_wch\(\)](#).

NAME

wgetch — get a single-byte character from the terminal

SYNOPSIS

```
#include <curses.h>

int wgetch(WINDOW *win);
```

DESCRIPTION

Refer to [getch\(\)](#).

NAME

wgetn_wstr, wget_wstr — get an array of wide characters and function key codes from a terminal

SYNOPSIS

```
EC #include <curses.h>
int wgetn_wstr(WINDOW *win, wint_t *wstr, int n);
int wget_wstr(WINDOW *win, wint_t *wstr);
```

DESCRIPTION

Refer to [getn_wstr\(\)](#).

NAME

wgetnstr, wgetstr — get a multi-byte character string from the terminal

SYNOPSIS

```
#include <curses.h>
```

```
EC int wgetnstr(WINDOW *win, char *str, int n);  
   int wgetstr(WINDOW *win, char *str);
```

DESCRIPTION

Refer to *getnstr()*.

NAME

whline, wvline — draw lines from single-byte characters and renditions

SYNOPSIS

```
EC #include <curses.h>
    int whline(WINDOW *win, chtype ch, int n);
    int wvline(WINDOW *win, chtype ch, int n);
```

DESCRIPTION

Refer to [hline\(\)](#).

NAME

whline_set, wvline_set — draw lines from complex characters and renditions

SYNOPSIS

```
EC #include <curses.h>
int whline_set(WINDOW *win, const cchar_t *wch, int n);
int wvline_set(WINDOW *win, const cchar_t *wch, int n);
```

DESCRIPTION

Refer to [hline_set\(\)](#).

NAME

win_wch — extract a complex character and rendition from a window

SYNOPSIS

```
EC #include <curses.h>
    int win_wch(WINDOW *win, cchar_t *wcval);
```

DESCRIPTION

Refer to [in_wch\(\)](#).

NAME

win_wchnstr, win_wchstr — extract an array of complex characters and renditions from a window

SYNOPSIS

```
EC #include <curses.h>

int win_wchnstr(WINDOW *win, cchar_t *wchstr, int n);
int win_wchstr(WINDOW *win, cchar_t *wchstr);
```

DESCRIPTION

Refer to [in_wchnstr\(\)](#).

NAME

winch — input a single-byte character and rendition from a window

SYNOPSIS

```
#include <curses.h>

chtype winch(WINDOW *win);
```

DESCRIPTION

Refer to *inch()*.

NAME

winchnstr, winchstr — input an array of single-byte characters and renditions from a window

SYNOPSIS

```
EC #include <curses.h>
    int winchnstr(WINDOW *win, chtype *chstr, int n);
    int winchstr(WINDOW *win, chtype *chstr);
```

DESCRIPTION

Refer to *inchnstr()*.

NAME

winnstr, winstr — input a multi-byte character string from a window

SYNOPSIS

```
EC #include <curses.h>
    int winnstr(WINDOW *win, char *str, int n);
    int winstr(WINDOW *win, char *str);
```

DESCRIPTION

Refer to *innstr()*.

NAME

winnwstr, winwstr — input a string of wide characters from a window

SYNOPSIS

```
EC #include <curses.h>
    int winnwstr(WINDOW *win, wchar_t *wstr, int n);
    int winwstr(WINDOW *win, wchar_t *wstr);
```

DESCRIPTION

Refer to *innwstr()*.

NAME

wins_nwstr, wins_wstr — insert a wide-character string into a window

SYNOPSIS

```
EC #include <curses.h>

int wins_nwstr(WINDOW *win, const wchar_t *wstr, int n);
int wins_wstr(WINDOW *win, const wchar_t *wstr);
```

DESCRIPTION

Refer to [ins_nwstr\(\)](#).

NAME

wins_wch — insert a complex character and rendition into a window

SYNOPSIS

```
EC #include <curses.h>
int wins_wch(WINDOW *win, const cchar_t *wch);
```

DESCRIPTION

Refer to [ins_wch\(\)](#).

NAME

winsch — insert a single-byte character and rendition into a window

SYNOPSIS

```
#include <curses.h>

int winsch(WINDOW *win, chtype ch);
```

DESCRIPTION

Refer to *insch()*.

NAME

winsdelln — delete or insert lines into a window

SYNOPSIS

```
EC #include <curses.h>
    int winsdelln(WINDOW *win, int n);
```

DESCRIPTION

Refer to *insdelln()*.

NAME

winsertln — insert lines into a window

SYNOPSIS

```
#include <curses.h>

int winsertln(WINDOW *win);
```

DESCRIPTION

Refer to [insertln\(\)](#).

NAME

winsnstr, winsstr — insert a multi-byte character string into a window

SYNOPSIS

```
EC #include <curses.h>
    int winsnstr(WINDOW *win, const char *str, int n);
    int winsstr(WINDOW *win, const char *str);
```

DESCRIPTION

Refer to *insnstr()*.

NAME

wmove — window cursor location functions

SYNOPSIS

```
#include <curses.h>

int wmove(WINDOW *win, int y, int x);
```

DESCRIPTION

Refer to *move()*.

NAME

wnoutrefresh, wrefresh — refresh windows and lines

SYNOPSIS

```
#include <curses.h>

int wnoutrefresh(WINDOW *win);
int wrefresh(WINDOW *win);
```

DESCRIPTION

Refer to *doupdate()*.

NAME

wprintw — print formatted output in window

SYNOPSIS

```
#include <curses.h>

int wprintw(WINDOW *win, const char *fmt, ...);
```

DESCRIPTION

Refer to *mvprintw()*.

NAME

wredrawln — line update status functions

SYNOPSIS

```
EC #include <curses.h>
int wredrawln(WINDOW *win, int beg_line, int num_lines);
```

DESCRIPTION

Refer to *redrawwin()*.

NAME

wscanw — convert formatted input from a window

SYNOPSIS

```
#include <curses.h>

int wscanw(WINDOW *win, const char *fmt, ...);
```

DESCRIPTION

Refer to [mvscanw\(\)](#).

NAME

wscrl — scroll a Curses window

SYNOPSIS

```
EC #include <curses.h>
    int wscrl(WINDOW *win, int n);
```

DESCRIPTION

Refer to *scrl()*.

NAME

wsetsrreg — terminal output control functions

SYNOPSIS

```
#include <curses.h>

int wsetsrreg(WINDOW *win, int top, int bot);
```

DESCRIPTION

Refer to *clearok()*.

NAME

wstandend, wstandout — set and clear window attributes

SYNOPSIS

```
#include <curses.h>

int wstandend(WINDOW *win);
int wstandout(WINDOW *win);
```

DESCRIPTION

Refer to *standend()*.

NAME

wsyncdown, wsyncup — synchronize a window with its parents or children

SYNOPSIS

```
EC    #include <curses.h>
      void wsyncdown(WINDOW *win);
      void wsyncup(WINDOW *win);
```

DESCRIPTION

Refer to [syncok\(\)](#).

NAME

wtimeout — control blocking on input

SYNOPSIS

```
EC #include <curses.h>
    void wtimeout(WINDOW *win, int delay);
```

DESCRIPTION

Refer to *notimeout()*.

NAME

wtouchln — window refresh control functions

SYNOPSIS

```
EC #include <curses.h>
int wtouchln(WINDOW *win, int y, int n, int changed);
```

DESCRIPTION

Refer to [is_linetouched\(\)](#).

NAME

wunctrl — generate printable representation of a wide character

SYNOPSIS

```
EC #include <curses.h>
wchar_t *wunctrl(cchar_t *wc);
```

DESCRIPTION

The *wunctrl()* function generates a wide-character string that is a printable representation of the wide character *wc*.

This function also performs the following processing on the input argument:

- Control characters are converted to the 'X' notation.
- Any rendition information is removed.

RETURN VALUE

Upon successful completion, the *wunctrl()* function returns the generated string. Otherwise, it returns a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS


None.

SEE ALSO

keyname(), *unctrl()*, [<curses.h>](#)

CHANGE HISTORY

First released in Issue 4.



Chapter 5
Headers

This chapter describes the contents of headers used by the Curses functions, macros, and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros, and defined types. Each function in [Chapter 4](#) specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

NAME

curses.h — definitions for screen handling and optimization functions

SYNOPSIS

```
#include <curses.h>
```

DESCRIPTION**Objects**

EC The **<curses.h>** header provides a declaration for `COLOR_PAIRS`, `COLORS`, `COLS`, `curscr`, `LINES`, and `stdscr`.

Macros

The following macros are defined:

| | |
|------------------------------|--|
| EOF | Function return value for end-of-file, as described in <stdio.h> . |
| ERR | Function return value for failure. |
| FALSE | Boolean <i>false</i> value. |
| EC <code>KEY_CODE_YES</code> | Function return value indicating that a <code>wint_t</code> variable contains a key code. |
| OK | Function return value for success. |
| TRUE | Boolean <i>true</i> value. |
| WEOF | Wide-character function return value for end-of-file, as described in <wchar.h> . |

The following macro is defined:

EC `_XOPEN_CURSES` X/Open Enhanced Curses test macro.

Data Types

The following data type is defined as a macro:

`bool` As described in **<stdbool.h>**.

The following data types are defined through **typedef**:

| | |
|-------------------------|--|
| EC <code>attr_t</code> | An OR'ed set of attributes. |
| EC <code>chtype</code> | A character, attributes, and a color-pair. |
| <code>SCREEN</code> | An opaque terminal representation. |
| EC <code>wchar_t</code> | As described in <stddef.h> . |
| EC <code>wint_t</code> | As described in <wchar.h> . |
| EC <code>cchar_t</code> | References a string of wide characters. |
| <code>WINDOW</code> | An opaque window representation. |

These data types are described in more detail in [Section 2.3](#) (on page 12).

The inclusion of **<curses.h>** may make visible all symbols from the headers **<stdio.h>**, **<term.h>**, **<termios.h>**, and **<wchar.h>**.

Attribute Bits

EC The following macros are used to manipulate objects of type **attr_t**:

| | |
|---------------|--|
| WA_ALTCHARSET | Alternate character set |
| WA_BLINK | Blinking |
| WA_BOLD | Extra bright or bold |
| WA_DIM | Half bright |
| WA_HORIZONTAL | Horizontal highlight |
| WA_INVIS | Invisible |
| WA_LEFT | Left highlight |
| WA_LOW | Low highlight |
| WA_PROTECT | Protected |
| WA_REVERSE | Reverse video |
| WA_RIGHT | Right highlight |
| WA_STANDOUT | Best highlighting mode of the terminal |
| WA_TOP | Top highlight |
| WA_UNDERLINE | Underlining |
| WA_VERTICAL | Vertical highlight |

These attribute flags shall be distinct.

The following macros are used to manipulate attribute bits in objects of type **chtype**:

| | | |
|----|--------------|--|
| EC | A_ALTCHARSET | Alternate character set |
| | A_BLINK | Blinking |
| | A_BOLD | Extra bright or bold |
| | A_DIM | Half bright |
| EC | A_INVIS | Invisible |
| | A_PROTECT | Protected |
| | A_REVERSE | Reverse video |
| | A_STANDOUT | Best highlighting mode of the terminal |
| | A_UNDERLINE | Underlining |

EC These attribute flags need not be distinct except when `_XOPEN_CURSES` is defined.

The following macros can be used as bit-masks to extract the components of a **chtype**:

| | | |
|----|--------------|--|
| | A_ATTRIBUTES | Bit-mask to extract attributes |
| | A_CHARTEXT | Bit-mask to extract a character |
| EC | A_COLOR | Bit-mask to extract color-pair information |

Line-Drawing Macros

EC

The < curses.h > header defines the macros shown in the leftmost two columns of the following table for use in drawing lines. The macros that begin with ACS_ are char constants. The macros that begin with WACS_ are cchar_t constants used with the wide-character interfaces that take a pointer to a cchar_t.

In the POSIX locale, the characters shown in the POSIX Locale Default column are used when the terminal database does not specify a value using the acsc capability as described in Section A.1.12 (on page 361).

| char Constant | cchar_t Constant | POSIX Locale Default | Glyph Description |
|---------------|------------------|----------------------|-------------------------|
| ACS_ULCORNER | WACS_ULCORNER | + | upper left-hand corner |
| ACS_LLCORNER | WACS_LLCORNER | + | lower left-hand corner |
| ACS_URCORNER | WACS_URCORNER | + | upper right-hand corner |
| ACS_LRCORNER | WACS_LRCORNER | + | lower right-hand corner |
| ACS_RTEE | WACS_RTEE | + | right tee (⊖) |
| ACS_LTEE | WACS_LTEE | + | left tee (⊣) |
| ACS_BTEE | WACS_BTEE | + | bottom tee (⊥) |
| ACS_TTEE | WACS_TTEE | + | top tee (⊤) |
| ACS_HLINE | WACS_HLINE | - | horizontal line |
| ACS_VLINE | WACS_VLINE | | vertical line |
| ACS_PLUS | WACS_PLUS | + | plus |
| ACS_S1 | WACS_S1 | - | scan line 1 |
| ACS_S9 | WACS_S9 | - | scan line 9 |
| ACS_DIAMOND | WACS_DIAMOND | + | diamond |
| ACS_CKBOARD | WACS_CKBOARD | : | checker board (stipple) |
| ACS_DEGREE | WACS_DEGREE | ' | degree symbol |
| ACS_PLMINUS | WACS_PLMINUS | # | plus/minus |
| ACS_BULLET | WACS_BULLET | o | bullet |
| ACS_LARROW | WACS_LARROW | < | arrow pointing left |
| ACS_RARROW | WACS_RARROW | > | arrow pointing right |
| ACS_DARROW | WACS_DARROW | v | arrow pointing down |
| ACS_UARROW | WACS_UARROW | ^ | arrow pointing up |
| ACS_BOARD | WACS_BOARD | # | board of squares |
| ACS_LANTERN | WACS_LANTERN | # | lantern symbol |
| ACS_BLOCK | WACS_BLOCK | # | solid square block |

Color-Related Macros

EC The following color-related macros are defined:

```
COLOR_BLACK
COLOR_BLUE
COLOR_GREEN
COLOR_CYAN
COLOR_RED
COLOR_MAGENTA
COLOR_YELLOW
COLOR_WHITE
```

The following color-related macros are defined, and may also be declared as functions:

```
int COLOR_PAIR(int);
int PAIR_NUMBER(int);
```

Coordinate-Related Macros

The following coordinate-related macros are defined:

```
EC void getbegyx(WINDOW *win, int y, int x);
void getmaxyx(WINDOW *win, int y, int x);
void getparyx(WINDOW *win, int y, int x);
void getyx(WINDOW *win, int y, int x);
```

Key Codes

The following macros representing function key values are defined and have distinct values where each value is less than {CHAR_MIN} or greater then {UCHAR_MAX}.

| Key Code | Description |
|---------------|-----------------------|
| KEY_A1 | Upper left of keypad |
| KEY_A3 | Upper right of keypad |
| KEY_B2 | Center of keypad |
| KEY_BACKSPACE | Backspace |
| EC KEY_BEG | Beginning key |
| KEY_BREAK | Break key |
| EC KEY_BTAB | Back tab key |
| KEY_C1 | Lower left of keypad |
| KEY_C3 | Lower right of keypad |
| EC KEY_CANCEL | Cancel key |
| KEY_CATAB | Clear all tabs |
| KEY_CLEAR | Clear screen |
| EC KEY_CLOSE | Close key |
| KEY_COMMAND | Cmd (command) key |
| KEY_COPY | Copy key |
| KEY_CREATE | Create key |
| KEY_CTAB | Clear tab |

| | Key Code | Description |
|----|-------------------|--|
| | KEY_DC | Delete character |
| | KEY_DL | Delete line |
| | KEY_DOWN | Down arrow key |
| | KEY_EIC | Exit insert char mode |
| EC | KEY_END | End key |
| | KEY_ENTER | Enter or send |
| | KEY_EOL | Clear to end of line |
| | KEY_EOS | Clear to end of screen |
| EC | KEY_EXIT | Exit key |
| | KEY_F0 | Function keys; space for 64 keys is reserved |
| | KEY_F(<i>n</i>) | For $0 \leq n \leq 63$ |
| EC | KEY_FIND | Find key |
| | KEY_HELP | Help key |
| | KEY_HOME | Home key |
| | KEY_IC | Insert char or enter insert mode |
| | KEY_IL | Insert line |
| | KEY_LEFT | Left arrow key |
| | KEY_LL | Home down or bottom |
| EC | KEY_MARK | Mark key |
| | KEY_MESSAGE | Message key |
| | KEY_MOVE | Move key |
| | KEY_NEXT | Next object key |
| | KEY_NPAGE | Next page |
| EC | KEY_OPEN | Open key |
| | KEY_OPTIONS | Options key |
| | KEY_PPAGE | Previous page |
| EC | KEY_PREVIOUS | Previous object key |
| | KEY_PRINT | Print or copy |
| EC | KEY_REDO | Redo key |
| | KEY_REFERENCE | Reference key |
| | KEY_REFRESH | Refresh key |
| | KEY_REPLACE | Replace key |
| | KEY_RESET | Reset or hard reset |
| EC | KEY_RESTART | Restart key |
| | KEY_RESUME | Resume key |
| | KEY_RIGHT | Right arrow key |
| EC | KEY_SAVE | Save key |
| | KEY_SBEG | Shifted beginning key |
| | KEY_SCANCEL | Shifted cancel key |
| | KEY_SCOMMAND | Shifted command key |
| | KEY_SCOPY | Shifted copy key |
| | KEY_SCREATE | Shifted create key |
| | KEY_SDC | Shifted delete char key |
| | KEY_SDL | Shifted delete line key |
| | KEY_SELECT | Select key |
| | KEY_SEND | Shifted end key |
| | KEY_SEOL | Shifted clear line key |
| | KEY_SEXIT | Shifted exit key |
| | KEY_SF | Scroll 1 line forward |

| | Key Code | Description |
|----|---------------|----------------------------------|
| EC | KEY_SFIND | Shifted find key |
| | KEY_SHELP | Shifted help key |
| | KEY_SHOME | Shifted home key |
| | KEY_SIC | Shifted input key |
| | KEY_SLEFT | Shifted left arrow key |
| | KEY_SMESSAGE | Shifted message key |
| | KEY_SMOVE | Shifted move key |
| | KEY_SNEXT | Shifted next key |
| | KEY_SOPTIONS | Shifted options key |
| | KEY_SPREVIOUS | Shifted prev key |
| | KEY_SPRINT | Shifted print key |
| | KEY_SR | Scroll 1 line backward (reverse) |
| EC | KEY_SREDO | Shifted redo key |
| | KEY_SREPLACE | Shifted replace key |
| | KEY_SRESET | Soft (partial) reset |
| EC | KEY_SRIGHT | Shifted right arrow |
| | KEY_SRSUME | Shifted resume key |
| | KEY_SSAVE | Shifted save key |
| | KEY_SSUSPEND | Shifted suspend key |
| | KEY_STAB | Set tab |
| EC | KEY_SUNDO | Shifted undo key |
| | KEY_SUSPEND | Suspend key |
| | KEY_UNDO | Undo key |
| | KEY_UP | Up arrow key |

The virtual keypad is a 3-by-3 keypad arranged as follows:

| | | |
|------|------|-------|
| A1 | UP | A3 |
| LEFT | B2 | RIGHT |
| C1 | DOWN | C3 |

Each legend, such as A1, corresponds to a macro for a key code from the preceding table, such as KEY_A1.

Function Prototypes

The following are declared as functions, and may also be defined as macros:

| | | |
|----|-----|------------------------------------|
| | int | addch(const chtype); |
| EC | int | addchnstr(const chtype *, int); |
| | int | addchstr(const chtype *); |
| EC | int | addnstr(const char *, int); |
| | int | addnwstr(const wchar_t *, int); |
| | int | addstr(const char *); |
| | int | add_wch(const cchar_t *); |
| | int | add_wchnstr(const cchar_t *, int); |
| | int | add_wchstr(const cchar_t *); |
| | int | addwstr(const wchar_t *); |
| | int | attroff(int); |
| | int | attron(int); |
| | int | attrset(int); |

```

EC   int      attr_get(attr_t *, short *, void *);
int   attr_off(attr_t, void *);
int   attr_on(attr_t, void *);
int   attr_set(attr_t, short, void *);
int   baudrate(void);
int   beep(void);
EC   int      bkgd(chtype);
void   bkgdset(chtype);
int   bkgrnd(const cchar_t *);
void   bkgrndset(const cchar_t *);
int   border(chtype, chtype, chtype, chtype, chtype, chtype,
           chtype, chtype);
int   border_set(const cchar_t *, const cchar_t *,
           const cchar_t *, const cchar_t *, const cchar_t *,
           const cchar_t *, const cchar_t *);
int   box(WINDOW *, chtype, chtype);
EC   int      box_set(WINDOW *, const cchar_t *, const cchar_t *);
bool   can_change_color(void);
int   cbreak(void);
EC   int      chgat(int, attr_t, short, const void *);
int   clearok(WINDOW *, bool);
int   clear(void);
int   clrtoobot(void);
int   clrtoeol(void);
EC   int      color_content(short, short *, short *, short *);
int   color_set(short, void *);
int   copywin(const WINDOW *, WINDOW *, int, int, int, int, int,
           int, int);
int   curs_set(int);
int   def_prog_mode(void);
int   def_shell_mode(void);
int   delay_output(int);
int   delch(void);
int   deleteln(void);
EC   void     delscreen(SCREEN *);
int   delwin(WINDOW *);
EC   WINDOW  *derwin(WINDOW *, int, int, int, int);
int   doupdate(void);
EC   WINDOW  *dupwin(WINDOW *);
int   echo(void);
EC   int      echochar(const chtype);
int   echo_wchar(const cchar_t *);
int   endwin(void);
char  erasechar(void);
int   erase(void);
EC   int      erasewchar(wchar_t *);
void   filter(void);
int   flash(void);
int   flushinp(void);
EC   chtype   getbkgd(WINDOW *);
int   getbkgrnd(cchar_t *);
int   getcchar(const cchar_t *, wchar_t *, attr_t *, short *,

```

```

        void *);
int      getch(void);
EC int    getnstr(char *, int);
int      getn_wstr(wint_t *, int);
int      getstr(char *);
EC int    get_wch(wint_t *);
WINDOW  *getwin(FILE *);
int      get_wstr(wint_t *);
int      halfdelay(int);
bool     has_colors(void);
bool     has_ic(void);
bool     has_il(void);
EC int    hline(chtype, int);
int      hline_set(const cchar_t *, int);
void     idcok(WINDOW *, bool);
int      idlok(WINDOW *, bool);
EC void   immedok(WINDOW *, bool);
chtype   inch(void);
EC int    inchnstr(chtype *, int);
int      inchstr(chtype *);
WINDOW  *initscr(void);
EC int    init_color(short, short, short, short);
int      init_pair(short, short, short);
int      innstr(char *, int);
int      innwstr(wchar_t *, int);
int      insch(chtype);
EC int    insdelln(int);
int      insertln(void);
EC int    insnstr(const char *, int);
int      ins_nwstr(const wchar_t *, int);
int      insstr(const char *);
int      instr(char *);
int      ins_wch(const cchar_t *);
int      ins_wstr(const wchar_t *);
int      intrflush(WINDOW *, bool);
EC int    in_wch(cchar_t *);
int      in_wchnstr(cchar_t *, int);
int      in_wchstr(cchar_t *);
int      inwstr(wchar_t *);
bool     isendwin(void);
bool     is_linetouched(WINDOW *, int);
bool     is_wintouched(WINDOW *);
char     *keyname(int);
char     *key_name(wchar_t);
int      keypad(WINDOW *, bool);
char     killchar(void);
EC int    killwchar(wchar_t *);
int      leaveok(WINDOW *, bool);
char     *longname(void);
EC int    meta(WINDOW *, bool);
int      move(int, int);
int      mvaddch(int, int, const chtype);

```

```

EC    int    mvaddchnstr(int, int, const chtype *, int);
int    mvaddchstr(int, int, const chtype *);
EC    int    mvaddnstr(int, int, const char *, int);
int    mvaddnwstr(int, int, const wchar_t *, int);
int    mvaddstr(int, int, const char *);
int    mvadd_wch(int, int, const cchar_t *);
int    mvadd_wchnstr(int, int, const cchar_t *, int);
int    mvadd_wchstr(int, int, const cchar_t *);
int    mvaddwstr(int, int, const wchar_t *);
int    mvchgat(int, int, int, attr_t, short, const void *);
int    mvcur(int, int, int, int);
int    mvdelch(int, int);
EC    int    mvderwin(WINDOW *, int, int);
int    mvgetch(int, int);
EC    int    mvgetnstr(int, int, char *, int);
int    mvgetn_wstr(int, int, wint_t *, int);
int    mvgetstr(int, int, char *);
EC    int    mvget_wch(int, int, wint_t *);
int    mvget_wstr(int, int, wint_t *);
int    mvhline(int, int, chtype, int);
int    mvhline_set(int, int, const cchar_t *, int);
chtype mvinch(int, int);
EC    int    mvinchnstr(int, int, chtype *, int);
int    mvinchstr(int, int, chtype *);
int    mvinnstr(int, int, char *, int);
int    mvinnwstr(int, int, wchar_t *, int);
int    mvinsch(int, int, chtype);
EC    int    mvinsnstr(int, int, const char *, int);
int    mvins_nwstr(int, int, const wchar_t *, int);
int    mvinsstr(int, int, const char *);
int    mvinstr(int, int, char *);
int    mvins_wch(int, int, const cchar_t *);
int    mvins_wstr(int, int, const wchar_t *);
int    mvin_wch(int, int, cchar_t *);
int    mvin_wchnstr(int, int, cchar_t *, int);
int    mvin_wchstr(int, int, cchar_t *);
int    mvinwstr(int, int, wchar_t *);
int    mvprintw(int, int, const char *, ...);
int    mvscanw(int, int, const char *, ...);
EC    int    mvvline(int, int, chtype, int);
int    mvvline_set(int, int, const cchar_t *, int);
int    mwaddch(WINDOW *, int, int, const chtype);
EC    int    mwaddchnstr(WINDOW *, int, int, const chtype *, int);
int    mwaddchstr(WINDOW *, int, int, const chtype *);
EC    int    mwaddnstr(WINDOW *, int, int, const char *, int);
int    mwaddnwstr(WINDOW *, int, int, const wchar_t *, int);
int    mwaddstr(WINDOW *, int, int, const char *);
int    mwadd_wch(WINDOW *, int, int, const cchar_t *);
int    mwadd_wchnstr(WINDOW *, int, int, const cchar_t *, int);
int    mwadd_wchstr(WINDOW *, int, int, const cchar_t *);
int    mwaddwstr(WINDOW *, int, int, const wchar_t *);
int    mwchgat(WINDOW *, int, int, int, attr_t, short,

```



```

                                const void *);
int      mvwdelch(WINDOW *, int, int);
int      mvwgetch(WINDOW *, int, int);
EC      int      mvwgetnstr(WINDOW *, int, int, char *, int);
int      mvwgetn_wstr(WINDOW *, int, int, wint_t *, int);
int      mvwgetstr(WINDOW *, int, int, char *);
EC      int      mvwget_wch(WINDOW *, int, int, wint_t *);
int      mvwget_wstr(WINDOW *, int, int, wint_t *);
int      mvwhline(WINDOW *, int, int, chtype, int);
int      mvwhline_set(WINDOW *, int, int, const cchar_t *, int);
int      mvwin(WINDOW *, int, int);
chtype   mvwinch(WINDOW *, int, int);
EC      int      mvwinchnstr(WINDOW *, int, int, chtype *, int);
int      mvwinchstr(WINDOW *, int, int, chtype *);
int      mvwinnstr(WINDOW *, int, int, char *, int);
int      mvwinnwstr(WINDOW *, int, int, wchar_t *, int);
int      mvwinsch(WINDOW *, int, int, chtype);
EC      int      mvwinsnstr(WINDOW *, int, int, const char *, int);
int      mvwins_nwstr(WINDOW *, int, int, const wchar_t *, int);
int      mvwinsstr(WINDOW *, int, int, const char *);
int      mvwinstr(WINDOW *, int, int, char *);
int      mvwins_wch(WINDOW *, int, int, const cchar_t *);
int      mvwins_wstr(WINDOW *, int, int, const wchar_t *);
int      mvwin_wch(WINDOW *, int, int, cchar_t *);
int      mvwin_wchnstr(WINDOW *, int, int, cchar_t *, int);
int      mvwin_wchstr(WINDOW *, int, int, cchar_t *);
int      mvwinwstr(WINDOW *, int, int, wchar_t *);
int      mvwprintw(WINDOW *, int, int, const char *, ...);
int      mvwscanw(WINDOW *, int, int, const char *, ...);
EC      int      mvwvline(WINDOW *, int, int, chtype, int);
int      mvwvline_set(WINDOW *, int, int, const cchar_t *, int);
int      napms(int);
WINDOW   *newpad(int, int);
SCREEN   *newterm(const char *, FILE *, FILE *);
WINDOW   *newwin(int, int, int, int);
int      nl(void);
int      nocbreak(void);
int      nodelay(WINDOW *, bool);
int      noecho(void);
int      nonl(void);
EC      void     noqiflush(void);
int      noraw(void);
EC      int      notimeout(WINDOW *, bool);
int      overlay(const WINDOW *, WINDOW *);
int      overwrite(const WINDOW *, WINDOW *);
EC      int      pair_content(short, short *, short *);
int      pechochar(WINDOW *, chtype);
int      pecho_wchar(WINDOW *, const cchar_t*);
int      pnoutrefresh(WINDOW *, int, int, int, int, int, int);
int      prefresh(WINDOW *, int, int, int, int, int, int);
int      printw(const char *, ...);

```

```

EC    int      putp(const char *);
      int      putwin(WINDOW *, FILE *);
      void     qiflush(void);
      int      raw(void);
EC    int      redrawwin(WINDOW *);
      int      refresh(void);
      int      reset_prog_mode(void);
      int      reset_shell_mode(void);
      int      resetty(void);
EC    int      ripoffline(int, int (*)(WINDOW *, int));
      int      savetty(void);
      int      scanw(const char *, ...);
EC    int      scr_dump(const char *);
      int      scr_init(const char *);
      int      scl(int);
      int      scroll(WINDOW *);
      int      scrollok(WINDOW *, bool);
EC    int      scr_restore(const char *);
      int      scr_set(const char *);
      int      setcchar(cchar_t*, const wchar_t*, const attr_t, short,
                       const void*);
      int      setscrreg(int, int);
      SCREEN  *set_term(SCREEN *);
      int      setupterm(char *, int, int *);
EC    int      slk_attr_off(const attr_t, void *);
      int      slk_attron(const chtype);
      int      slk_attr_on(const attr_t, void *);
      int      slk_attrset(const attr_t, short, void *);
      int      slk_attrset(const chtype);
      int      slk_clear(void);
      int      slk_color(short);
      int      slk_init(int);
      char     *slk_label(int);
      int      slk_noutrefresh(void);
      int      slk_refresh(void);
      int      slk_restore(void);
      int      slk_set(int, const char *, int);
      int      slk_touch(void);
      int      slk_wset(int, const wchar_t *, int);
      int      standend(void);
      int      standout(void);
EC    int      start_color(void);
      WINDOW  *subpad(WINDOW *, int, int, int, int);
      WINDOW  *subwin(WINDOW *, int, int, int, int);
EC    int      syncok(WINDOW *, bool);
      chtype  termattrs(void);
      attr_t  term_attrs(void);
      char     *termname(void);
      int      tigetflag(const char *);
      int      tigetnum(const char *);
      char     *tigetstr(const char *);

```

```

void      timeout(int);
int       touchline(WINDOW *, int, int);
int       touchwin(WINDOW *);
EC char   *tiparm(const char *, ...);
EC OB char *tparm(const char *, long, long, long, long, long, long,
                long, long, long);
int       typeahead(int);
EC int    ungetch(int);
int       unget_wch(const wchar_t);
int       untouchwin(WINDOW *);
void      use_env(bool);
int       vid_attr(attr_t, short, void *);
int       vidattr(chtype);
int       vid_puts(attr_t, short, void *, int (*)(int));
int       vidputs(chtype, int (*)(int));
int       vline(chtype, int);
int       vline_set(const cchar_t *, int);
int       vw_printw(WINDOW *, const char *, va_list);
int       vw_scanw(WINDOW *, const char *, va_list);
int       waddch(WINDOW *, const chtype);
EC int    waddchnstr(WINDOW *, const chtype *, int);
int       waddchstr(WINDOW *, const chtype *);
EC int    waddnstr(WINDOW *, const char *, int);
int       waddnwstr(WINDOW *, const wchar_t *, int);
int       waddstr(WINDOW *, const char *);
int       wadd_wch(WINDOW *, const cchar_t *);
int       wadd_wchnstr(WINDOW *, const cchar_t *, int);
int       wadd_wchstr(WINDOW *, const cchar_t *);
int       waddwstr(WINDOW *, const wchar_t *);
int       wattroff(WINDOW *, int);
int       wattron(WINDOW *, int);
int       wattrset(WINDOW *, int);
EC int    wattr_get(WINDOW *, attr_t *, short *, void *);
int       wattr_off(WINDOW *, attr_t, void *);
int       wattr_on(WINDOW *, attr_t, void *);
int       wattr_set(WINDOW *, attr_t, short, void *);
int       wbkgd(WINDOW *, chtype);
void      wbkgdset(WINDOW *, chtype);
int       wbkgrnd(WINDOW *, const cchar_t *);
void      wbkgrndset(WINDOW *, const cchar_t *);
int       wborder(WINDOW *, chtype, chtype, chtype, chtype, chtype,
                chtype, chtype, chtype);
int       wborder_set(WINDOW *, const cchar_t *, const cchar_t *,
                const cchar_t *, const cchar_t *,
                const cchar_t *, const cchar_t *, const cchar_t *);
int       wchgat(WINDOW *, int, attr_t, short, const void *);
int       wclear(WINDOW *);
int       wclrtoobot(WINDOW *);
int       wclrtoeol(WINDOW *);
EC void   wcursyncup(WINDOW *);
int       wcolor_set(WINDOW *, short, void *);
int       wdelch(WINDOW *);

```

```

int      wdeleteln(WINDOW *);
EC      int      wechochar(WINDOW *, const chtype);
int      wecho_wchar(WINDOW *, const cchar_t *);
int      werase(WINDOW *);
EC      int      wgetbkgrnd(WINDOW *, cchar_t *);
int      wgetch(WINDOW *);
EC      int      wgetnstr(WINDOW *, char *, int);
int      wgetn_wstr(WINDOW *, wint_t *, int);
int      wgetstr(WINDOW *, char *);
EC      int      wget_wch(WINDOW *, wint_t *);
int      wget_wstr(WINDOW *, wint_t *);
int      whline(WINDOW *, chtype, int);
int      whline_set(WINDOW *, const cchar_t *, int);
chtype   winch(WINDOW *);
EC      int      winchnstr(WINDOW *, chtype *, int);
int      winchstr(WINDOW *, chtype *);
int      winnstr(WINDOW *, char *, int);
int      winnwstr(WINDOW *, wchar_t *, int);
int      winsch(WINDOW *, chtype);
EC      int      winsdelln(WINDOW *, int);
int      winsertln(WINDOW *);
EC      int      winsnstr(WINDOW *, const char *, int);
int      wins_nwstr(WINDOW *, const wchar_t *, int);
int      winsstr(WINDOW *, const char *);
int      winstr(WINDOW *, char *);
int      wins_wch(WINDOW *, const cchar_t *);
int      wins_wstr(WINDOW *, const wchar_t *);
int      win_wch(WINDOW *, cchar_t *);
int      win_wchnstr(WINDOW *, cchar_t *, int);
int      win_wchstr(WINDOW *, cchar_t *);
int      winwstr(WINDOW *, wchar_t *);
int      wmove(WINDOW *, int, int);
int      wnoutrefresh(WINDOW *);
int      wprintw(WINDOW *, const char *, ...);
EC      int      wredrawln(WINDOW *, int, int);
int      wrefresh(WINDOW *);
int      wscanw(WINDOW *, const char *, ...);
EC      int      wscrln(WINDOW *, int);
int      wsetscrreg(WINDOW *, int, int);
int      wstandend(WINDOW *);
int      wstandout(WINDOW *);
EC      void     wsyncup(WINDOW *);
void     wsyncdown(WINDOW *);
void     wtimeout(WINDOW *, int);
int      wtouchln(WINDOW *, int, int, int);
wchar_t  *wunctrl(cchar_t *);
int      wvline(WINDOW *, chtype, int);
int      wvline_set(WINDOW *, const cchar_t *, int);

```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Chapter 1, <stdbool.h> (in the XBD specification), <stdio.h> (in the XBD specification), <term.h>, <termios.h> (in the XBD specification), <unctrl.h>, <wchar.h> (in the XBD specification)

CHANGE HISTORY

First released in Issue 2.

Issue 4

The entry is completely rewritten to include new constants, data types, and function prototypes.

Issue 4, Version 2

This entry is completely rewritten to correct the function prototypes.

Issue 7

The prototypes for the following functions are updated:

mvscanw(), *mvwscanw()*, *newterm()*, *scanw()*, *tigetflag()*, *tigetnum()*, *tigetstr()*, *tparm()*, *vw_printw()*, *vw_scanw()*, *wscanw()*

The *tparm()* function has been marked obsolescent.

The *tiparm()* function has been added.

Corrigendum U018/3 is applied, adding the value of `_XOPEN_SOURCE` for environments that support the Base Specifications, Issue 5.

Corrigendum U018/5 is applied, correcting the *vw_printw()* function prototype.

Corrigendum U022/1 is applied, correcting the shading on the *addchnstr()* and *addchstr()* function prototypes.

Corrigendum U056/2 is applied, adding the value of `_XOPEN_SOURCE` for environments that support the Base Specifications, Issue 6.

Corrigendum U058/1 is applied, moving the *COLOR_PAIR()* and *PAIR_NUMBER()* functions prototypes into the "Color-Related Macros" section.

NAME

term.h — terminal capabilities

SYNOPSISEC `#include <term.h>`**DESCRIPTION**

The following data type is defined through **typedef**:

TERMINAL An opaque representation of the capabilities for a single terminal from the **terminfo** database.

The **<term.h>** header provides a declaration for the following object: *cur_term*. It represents the current terminal record from the **terminfo** database that the application has selected by calling *set_curterm()*.

The **<term.h>** header defines the variable names listed in the **Variable** column in the table in [Section 7.1.3](#) (on page 340).

The following are declared as functions, and may also be defined as macros:

```
int      del_curterm(TERMINAL *);
int      putp(const char *);
int      restartterm(char *, int, int *);
TERMINAL *set_curterm(TERMINAL *);
int      setupterm(char *, int, int *);
int      tigetflag(const char *);
int      tigetnum(const char *);
char     *tigetstr(const char *);
char     *tiparm(const char *, ...);
OB char  *tparm(const char *, long, long, long, long, long, long, long,
               long, long);
int      tputs(const char *, int, int (*)(int));
```

The **<term.h>** header defines the following data type as a macro:

```
bool     As described in <stdbool.h>.
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Chapter 7](#) (on page 337), *printf()*, (in the **XSH** specification), *putp()*, *tigetflag()*, **<stdbool.h>** (in the **XBD** specification)

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

This entry is corrected.

NAME

unctrl.h — definitions for *unctrl()*

SYNOPSIS

```
#include <unctrl.h>
```

DESCRIPTION

The <unctrl.h> header defines the **chtype** type as defined in <curses.h>.

The following is declared as a function, and may also be defined as a macro:

```
char *unctrl(chtype);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS


None.

SEE ALSO

unctrl(), <curses.h>

CHANGE HISTORY

First released in Issue 4.



Chapter 6
Utilities

This chapter describes the Curses utilities to support applications portability and consistency of user experience at the shell command level.

The Curses utilities shall conform to the requirements stated in the **XCU** specification, Section 1.4, Utility Description Defaults, as if the text in the **XCU** specification, Section 1.4, Utility Description Defaults contained the phrase “POSIX.1-2008 or XCurses, Issue 7” instead of “POSIX.1-2008”, and contained the phrase “the Curses utilities” instead of “the standard utilities”.

The Curses utilities shall conform completely to the utility syntax guidelines defined in the **XBD** specification, Section 12.2, Utility Syntax Guidelines, as if those guidelines contained the term “shall” instead of “should”. On some implementations, the utilities accept usage in violation of those guidelines for backwards-compatibility as well as accepting the required form.

If a Curses utility uses operands to represent files, it is implementation-defined whether the operand ‘-’ stands for standard input (or standard output) or for a file named -.

NAME

infocmp — compare or print out **terminfo** descriptions

SYNOPSIS

```
EC infocmp [-I|-L] [-1] [-A directory] [-B directory]
    [-s sortorder] [-w width] [termname]

infocmp -u [-I|-L] [-1] [-A directory] [-B directory]
    [-s sortorder] [-w width] termname termname...

infocmp [-c|-d|-n] [-A directory] [-B directory]
    [-s sortorder] [-w width] termname termname

infocmp -n [-A directory] [-B directory]
    [-s sortorder] [-w width]
```

DESCRIPTION

The *infocmp* utility compares a compiled **terminfo** entry with other **terminfo** entries, rewrites a **terminfo** description to take advantage of the **use= terminfo** field, or prints out a **terminfo** description from the compiled entry in a variety of formats.

It displays boolean fields first, then numeric fields, followed by the string fields.

If none of the **-I**, **-L**, or **-n** options are specified and zero or one *termname* is specified, the **-I** option is assumed. If none of the **-c**, **-d**, **-n**, or **-u** options are specified and two *termname* operands are specified, the **-d** option is assumed. If the **-u** option is not specified and more than two *termname* operands are specified, it is unspecified whether the **-u** option is assumed.

OPTIONS

The **-d**, **-c**, and **-n** options can be used for comparisons. The *infocmp* utility compares the **terminfo** description of the first terminal *termname* with each of the descriptions given by the entries for the other terminal's *termname*. If a capability is defined for only one of the terminals, the value returned will depend on the type of the capability: *F* for boolean variables, *-1* for integer variables, and a null string for string variables.

- d** Produce a list of each capability that is different between two entries. This option is useful to show the difference between two entries, created by different people, for the same or similar terminals.
- c** Produce a list of each capability that is common between two entries. Capabilities that are not set are ignored. This option can be used as a quick check to see if the **-u** option is worth using.
- n** Produce a list of each capability that is in neither entry. If no *termname* is given, the environment variable *TERM* will be used for both of the *termnames*. This can be used as a quick check to see if anything was left out of a description.

The **-I** and **-L** options will produce a source listing for the terminal named by the *termname* operand, or for the terminal named by the environment variable *TERM* if no *termname* operand is specified.

- I** Use the **terminfo** names.
- L** Use the long C variable name listed in **<term.h>**.
- u** Produce a **terminfo** source description of the first terminal *termname* which is relative to the sum of the descriptions given by the entries for the other terminals' *termnames*. It does this by analyzing the differences between the first *termname* and the other *termnames* and producing a description with **use=** fields for the other terminals. In this manner, it is possible to retrofit generic **terminfo** entries into a

terminal's description. Or, if two similar terminals exist, but were coded at different times, or by different people so that each description is a full description, using *infocmp* will show what can be done to change one description to be relative to the other.

A capability is displayed with an at-sign ('@ ') if it no longer exists in the first *termname*, but one of the other *termname* entries contains a value for it. A capability's value is displayed if the value in the first *termname* is not found in any of the other *termname* entries, or if the first of the other *termname* entries that has this capability gives a different value for that capability.

The order of the other *termname* entries is significant. Since the **terminfo** compiler **tic** does a left-to-right scan of the capabilities, specifying two **use=** entries that contain differing entries for the same capabilities will produce different results, depending on the order in which the entries are given. The *infocmp* utility will flag any such inconsistencies between the other *termname* entries as they are found.

Alternatively, specifying a capability after a **use=** entry that contains that capability will cause the second specification to be ignored. Using *infocmp* to recreate a description can be a useful check to make sure that everything was specified correctly in the original source description.

Another error that does not cause incorrect compiled files, but will slow down the compilation time, is specifying superfluous **use=** fields. The *infocmp* utility will flag any superfluous **use=** fields.

-s sortorder Sort the fields within each type according to the *sortorder* option-argument below:

d Leave fields in the order that they are stored in the **terminfo** database.

i Sort by **terminfo** name.

l Sort by the long C variable name.

If the **-s** option is not given, the fields are sorted alphabetically by the **terminfo** name within each type, except in the case of the **-L** option, which causes the sorting to be done by the long C variable name.

-1 Print the fields one to a line. Otherwise, the fields are printed several to a line to a maximum width of 60 characters.

-w width Change the output to *width* characters.

The location of the compiled **terminfo** database is taken from the environment variable *TERMINFO*. If the variable is not defined, or the terminal is not found in that location, the system **terminfo** database is used. The options **-A** and **-B** can be used to override this location.

-A directory Set *TERMINFO* for the first *termname*.

-B directory Set *TERMINFO* for the other *termnames*. With this, it is possible to compare descriptions for a terminal with the same name located in two different databases. This is useful for comparing descriptions for the same terminal created by different people.

OPERANDS

See the DESCRIPTION.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *infocmp*:

- LANG** Provide a default value for the internationalization variables that are unset or null. (See the **XBD** specification, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
- LC_ALL** If set to a non-empty string value, override the values of all the other internationalization variables.
- LC_CTYPE** Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
- LC_MESSAGES** Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
- NLSPATH** Determine the location of message catalogs for the processing of **LC_MESSAGES**.
- TERM** Determine the default terminal name. If this variable is unset or null, and no *termname* operand is specified, the behavior is unspecified.
- TERMINFO** Determine the location of a compiled **terminfo** database to be used instead of the system **terminfo** database.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **-I** or **-L** option is specified (explicitly or implicitly), the output shall consist of the **terminfo** source for the specified terminal in the format described in [Chapter 7](#) (on page 337), except that if the **-L** option is specified, the capabilities are identified by their long C variable names instead of the **Capname** short names defined in [Section 7.1.3](#) (on page 340).

When the **-d** option is specified (explicitly or implicitly), the output shall contain differences between the two entries in an unspecified format.

When the **-c** option is specified, the output shall contain a list of capabilities common between the two entries in an unspecified format.

When the **-n** option is specified, the output shall contain a list of capabilities that are in neither entry in an unspecified format.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

- 0 Successful completion.
- >0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

Implementations of *infocmp* exhibit different behavior when used outside the constraints of the SYNOPSIS. In particular, the behavior is unspecified when:

- The **-I** or **-L** option is used with more than one *termname* operand, without **-u**.
- The **-c**, **-d**, or **-n** option is used with one *termname* operand or with more than two *termname* operands.
- Any two or more of the **-I**, **-L**, **-c**, **-d**, and **-n** options are used together.

FUTURE DIRECTIONS

None.

SEE ALSO

[Chapter 7](#), *tic*, *untic*, [<term.h>](#), the XBD specification: Section 8.2, Internationalization Variables

CHANGE HISTORY

First introduced in Issue 7. Derived from Solaris 7.

NAME

tic — translate **terminfo** files from source to compiled format

SYNOPSIS

```
EC tic [-c] file...
```

DESCRIPTION

The *tic* utility translates **terminfo** files from the source format into the compiled format.

If the *TERMINFO* environment variable is set, the results shall be placed there; otherwise, they shall be placed in the system **terminfo** database.

The *tic* utility compiles all **terminfo** descriptions in the file or files specified by the *file* operand. When the *tic* utility finds a **use=** field, it searches first the current file, then reads in the compiled entry from the system **terminfo** database to complete the entry. If the environment variable *TERMINFO* is set, that directory is searched instead of the system **terminfo** database.

The *tic* utility may impose limits on the size of compiled entries and on the length of the *name* field. The limit on the size of compiled entries, if any, shall be at least 4096 bytes. The limit on the length of the *name* field, if any, shall be at least 128 bytes. The *tic* utility shall support terminal names of at least 14 bytes. Users creating portable **terminfo** description files should not exceed these minimum limits

OPTIONS

-c Check the file for errors only. Errors in the **use=** field need not be detected.

OPERANDS

See the DESCRIPTION.

STDIN

The standard input shall be used if a file operand is '-' and the implementation treats the '-' as meaning standard input. Otherwise, the standard input shall not be used. See the INPUT FILES section.

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tic*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the **XBD** specification, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

TERMINFO Determine the location of a compiled **terminfo** database to be used instead of the system **terminfo** database.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

Compiled **terminfo** database entries in unspecified format are created.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

Some implementations of the *tic* utility report an error if no *file* operands are specified; other implementations read **terminfo** descriptions from standard input or from a default file such as *./terminfo.src* in this case. This standard allows the latter two behaviors as extensions, but conforming applications are required to supply one or more *file* operands.

FUTURE DIRECTIONS

None.

SEE ALSO

[Chapter 7](#), *infocmp*, *untic*, the **XBD** specification: Section 8.2, Internationalization Variables

CHANGE HISTORY

First introduced in Issue 7. Derived from Tru64 UNIX.

NAME

tput — initialize a terminal or query **terminfo** database

SYNOPSIS

```
EC tput [-T type] capname [parm...]
```

```
tput -S
```

DESCRIPTION

When XCURSES is supported, this description for the *tput* utility replaces that in the XCU specification.

The *tput* utility uses the **terminfo** database to make the values of terminal-dependent capabilities and information available to the shell (see *sh* in the XCU specification); to clear, initialize, or reset the terminal; or to return the long name of the requested terminal type. The *tput* utility outputs a string if the capability attribute (*capname*) is of type **string**, or an integer if the attribute is of type **integer**. If the attribute is of type **boolean**, *tput* simply sets the exit status (0 for TRUE if the terminal has the capability, 1 for FALSE if it does not), and produces no output.

OPTIONS

The following options are supported:

- T *type* Indicate the type of terminal. Normally this option is unnecessary, because the default is taken from the environment variable *TERM*. If -T is specified, then the environment variables *LINES* and *COLUMNS* and the layer size will not be referenced.
- S Allow more than one capability per invocation of *tput*. The capabilities must be passed to *tput* from the standard input instead of from the command line (see the EXAMPLES section). Only one *capname* is allowed per line. The -S option changes the meaning of the 0 and 1 boolean and string exit statuses (see the EXIT STATUS section).

OPERANDS

The following operands shall be supported:

capname Indicate the capability attribute from the **terminfo** database. See [Chapter 7](#) (on page 337) for a complete list of capabilities and the *capname* associated with each.

In addition, in the POSIX locale the following strings shall be supported as *capname* operands:

- clear* Display the clear-screen sequence.
- init* If the **terminfo** database is present and an entry for the user's terminal exists (see -T *type* above), the following shall occur:
 1. If present, the terminal's initialization strings shall be output (*is1*, *is2*, *is3*, *if*, *ipro*).
 2. Any delays (for instance, <newline>) specified in the entry shall be set in the terminal attributes (see the XBD specification, Chapter 11, General Terminal Interface).
 3. Tabs expansion shall be turned on or off according to the specification in the entry.

4. If tabs are not expanded, standard tabs shall be set (every 8 spaces).

If an entry does not contain the information needed for any of the four above activities, that activity shall be silently skipped.

| | |
|-----------------|---|
| <i>reset</i> | Instead of putting out initialization strings, the terminal's reset strings shall be output if present (<i>rs1i</i> , <i>rs2</i> , <i>rs3</i> , <i>rf</i>). If the reset strings are not present, but initialization strings are, the initialization strings shall be output. Otherwise, reset shall act identically to init . |
| <i>longname</i> | If the terminfo database is present and an entry for the user's terminal exists (see -T type above), then the long name of the terminal shall be output. The long name is the last name in the <i>name</i> field of the terminals' entry. |
| <i>parm</i> | If the attribute is a string that takes parameters, the argument <i>parm</i> will be instantiated into the string. |

STDIN

If the **-S** option is specified, lines are read from standard input and processed as if the contents of each line had been specified as a *capname* operand followed by zero or more *parm* operands on the command line, except for the exit status.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tput*:

| | |
|--------------------|--|
| <i>COLUMNS</i> | Override the system-selected horizontal screen size. See the XBD specification, Chapter 8, Environment Variables for valid values and results when it is unset or null. |
| <i>LANG</i> | Provide a default value for the internationalization variables that are unset or null. (See the XBD specification, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.) |
| <i>LC_ALL</i> | If set to a non-empty string value, override the values of all the other internationalization variables. |
| <i>LC_CTYPE</i> | Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments). |
| <i>LC_MESSAGES</i> | Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error. |
| <i>LINES</i> | Override the system-selected vertical screen size. See the XBD specification, Chapter 8, Environment Variables for valid values and results when it is unset or null. |
| <i>NLSPATH</i> | Determine the location of message catalogs for the processing of <i>LC_MESSAGES</i> . |
| <i>TERM</i> | Determine the terminal type. If this variable is unset or null, and if the -T option is not specified, an unspecified default terminal type shall be used. |

ASYNCHRONOUS EVENTS

Default.

STDOUT

See the DESCRIPTION.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

- 0
 - If *capname* is of type boolean and **-S** is not specified, indicates TRUE.
 - If *capname* is of type string and **-S** is not specified, indicates *capname* is defined for this terminal type.
 - If *capname* is of type boolean or string and **-S** is specified, indicates that all lines were successful.
 - *capname* is of type integer.
 - The requested string was written successfully.
 - 1
 - If *capname* is of type boolean and **-S** is not specified, indicates FALSE.
 - If *capname* is of type string and **-S** is not specified, indicates that *capname* is not defined for this terminal type.
 - 2 Usage error.
 - 3 No information is available about the specified terminal type.
 - 4 The specified operand is invalid.
 - 255 *capname* is a numeric variable that is not specified in the **terminfo** database.
- Any other value
An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES**Using the tput command**

This example initializes the terminal according to the type of terminal in the environment variable *TERM*:

```
tput init
```

The next example resets an AT&T 5620 terminal, overriding the type of terminal in the environment variable *TERM*:

```
tput -T 5620 reset
```

The following example outputs the sequence to move the cursor to row 0, column 0 (the upper left corner of the screen, usually known as the “home” cursor position):

```
tput cup 0 0
```

The next example sends the sequence to move the cursor to row 23, column 4:

```
tput cup 23 4
```

The next example outputs the clear-screen sequence for the current terminal:

```
tput clear
```

The next command outputs the number of columns for the current terminal:

```
tput cols
```

The following command outputs the number of columns for the 450 terminal:

```
tput -T 450 cols
```

The next example sets the shell variable *bold* to the begin standout mode sequence, and *offbold* to the end standout mode sequence, for the current terminal and then uses them in a prompt:

```
bold=$(tput smso)
if [ $? -ne 0 ]
then
    ...
fi
offbold=$(tput rmso)
if [ $? -ne 0 ]
then
    ...
fi
printf %s "${bold}Please type in your name: ${offbold}"
```

This example sets the exit status to indicate whether the current terminal is a hardcopy terminal:

```
tput hc
```

The next example prints the long name from the **terminfo** database for the type of terminal specified in the environment variable *TERM*:

```
tput longname
```

This last example shows *tput* processing several capabilities in one invocation. This example clears the screen, moves the cursor to position 10,10, and turns on bold (extra bright) mode. The list is terminated by an exclamation mark (‘!’) on a line by itself:

```
tput -S <<!
clear
cup 10 10
bold
!
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Chapter 7](#), the **XBD** specification: Section 8.2, Internationalization Variables; the **XCU** specification: *sh, stty, tabs*

CHANGE HISTORY

First introduced in Issue 7. Derived from Solaris 7.

NAME

untic — **terminfo** de-compiler

SYNOPSIS

```
EC  untic [-f file]  
    untic term
```

DESCRIPTION

The *untic* utility translates a **terminfo** file from the compiled format into the source format suitable for use by the *tic* utility. If the environment variable *TERMINFO* is set to a pathname, *untic* checks for a compiled **terminfo** description of the terminal under the path specified by *TERMINFO* before checking the system **terminfo** database. Otherwise, only the system **terminfo** database is checked.

Normally *untic* uses the terminal type obtained from the *TERM* environment variable. Using the *term* operand, however, the user can specify the terminal type used.

When the *-f* option is specified, the *file* option argument specifies the file used for translation.

The *untic* utility writes the de-compiled **terminfo** description result to standard output.

OPTIONS

-f file Specify the file to be used. This option bypasses the use of the *TERM* and *TERMINFO* environment variables.

OPERANDS

The following operand shall be supported:

term Indicate the type of terminal. If this operand is not present, the terminal is derived from the environment variable *TERM*.

STDIN

Not used.

INPUT FILES

The input file is a compiled **terminfo** database entry, either present in the system **terminfo** database or created by the *tic* utility.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *untic*:

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

TERM Determine the default terminal name. If this variable is unset or null, and no *term* operand is specified, behavior is unspecified.

TERMINFO Determine the location of a compiled **terminfo** database to be used instead of the system **terminfo** database.

ASYNCHRONOUS EVENTS

Default.

STDOUT

See the DESCRIPTION.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

[Chapter 7](#), *infocmp*, *tic*, the **XBD** specification: Section 8.2, Internationalization Variables

CHANGE HISTORY

First introduced in Issue 7. Derived from HPUX.

Terminfo Source Format (ENHANCED CURSES)

EC The requirements in this chapter are in effect only for implementations that claim Enhanced Curses compliance.

The **terminfo** database contains a description of the capabilities of a variety of devices, such as terminals and printers. Devices are described by specifying a set of capabilities, by quantifying certain aspects of the device, and by specifying character sequences that effect particular results.

This chapter specifies the format of **terminfo** source files.

The *tic* utility, described in [Chapter 6](#) (on page 323), accepts source files in the format specified in this chapter and can be used to enter information into the **terminfo** database. A valid **terminfo** entry describing a given model of terminal can be added to **terminfo** on any X/Open-compliant implementation to permit use of the same terminal model.

[Section 7.1](#) describes the syntax of **terminfo** source files. The grammar and lexical conventions appear in [Section 7.1.2](#) (on page 338). A list of all terminal capabilities defined by The Open Group appears in [Section 7.1.3](#) (on page 340). An example follows in [Section 7.1.4](#) (on page 349). [Section A.1](#) (on page 353) describes the specification of devices in general, such as video terminals. [Section A.2](#) (on page 366) describes the specification of printers.

The **terminfo** database is often used by screen-oriented applications such as *vi* and Curses programs, as well as by some utilities such as *ls* and *more*. This usage allows them to work with a variety of devices without changes to the programs.

7.1 Source File Syntax

Source files can use the ISO 8859-1:1987 codeset. The behavior when the source file is in another codeset is unspecified. Traditional practice has been to translate information from other codesets into the source file syntax.

terminfo source files consist of one or more device descriptions. Each description defines a mnemonic name for the terminal model. Each description consists of a header (beginning in column one) and one or more lines that list the features for that particular device. Every line in a **terminfo** source file must end in a comma. Every line in a **terminfo** source file except the header must be indented with one or more white spaces (either spaces or tabs).

Entries in **terminfo** source files consist of a number of comma-separated fields. White space after each comma is ignored. Embedded commas must be escaped by using a backslash. The following example shows the format of a **terminfo** source file:

```
alias1 | alias2 | . . . | aliasn | longname,  
<white space> am, lines #24,  
<white space> home=\\Eeh,
```

The first line, commonly referred to as the header line, must begin in column one and must contain at least two aliases separated by vertical bars. The last field in the header line must be the long name of the device and it may contain any string.

Alias names must be unique in the **terminfo** database and they must conform to filenaming conventions established by implementation-defined **terminfo** compilation utilities. Implementations will recognize alias names consisting only of characters from the portable filename character set except that implementations need not accept a first character of minus ('-'). For example, a typical restriction is that they cannot contain white space or slashes. There may be further constraints imposed on source file values by the implementation-defined **terminfo** compilation utilities. [Section A.4.1](#) (on page 377) provides conventions for choosing alias names.

Each capability in **terminfo** is of one of the following types:

- Boolean capabilities show that a device has or does not have a particular feature.
- Numeric capabilities quantify particular features of a device.
- String capabilities provide sequences that can be used to perform particular operations on devices.

Whenever possible, capability names are chosen to be the same as or similar to those specified by ISO/IEC 6429:1992. Semantics are also intended to match those of that standard.

All string capabilities may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the following tables, have names beginning with `key_`. These capabilities are defined in `<term.h>`.

7.1.1 Minimum Guaranteed Limits

All X/Open-compliant implementations support at least the following limits for the **terminfo** source file:

| Source File Characteristic | Minimum Guaranteed Value |
|---|------------------------------|
| Length of a line | 1 023 bytes |
| Length of a terminal alias | 14 bytes |
| Length of a terminal model name | 128 bytes |
| Width of a single field | 128 bytes |
| Length of a string value | 1 000 bytes |
| Length of a string representing a numeric value | 99 digits |
| Magnitude of a numeric value | 0 up to and including 32 767 |

An implementation may support higher limits than those specified above.

7.1.2 Formal Grammar

The grammar and lexical conventions in this section together describe the syntax for **terminfo** terminal descriptions within a **terminfo** source file. A terminal description that satisfies the requirements of this section will be accepted by all implementations.

```

descriptions : START_OF_HEADER_LINE1 rest_of_header_line feature_lines
              | descriptions START_OF_HEADER_LINE rest_of_header_line
              | feature_lines
              ;

rest_of_header_line : PIPE LONGNAME COMMA NEWLINE

```

1. An ALIAS that begins in column one. This is handled by the lexical analyzer.


```

    | aliases PIPE LONGNAME COMMA NEWLINE
    ;

feature_lines : start_feature_line rest_of_feature_line
    | feature_lines start_feature_line rest_of_feature_line
    ;

start_feature_line : START_FEATURE_LINE_BOOLEAN2
    | START_FEATURE_LINE_NUMERIC3
    | START_FEATURE_LINE_STRING4
    ;

rest_of_feature_line : features COMMA NEWLINE
    | COMMA NEWLINE
    ;

features : COMMA feature
    | features COMMA feature
    ;

aliases : PIPE ALIAS
    | aliases PIPE ALIAS
    ;

feature : BOOLEAN
    | NUMERIC
    | STRING
    ;

```

The lexical conventions for **terminfo** descriptions are as follows:

1. White space consists of the ' ' and <tab> character.
2. An ALIAS may contain any graph⁵ characters other than ', ', '/ ', and '| '.
3. A LONGNAME may contain any print⁶ characters other than ', ' and '| '.
4. A BOOLEAN feature may contain any print characters other than ', ', '=' , and '# '.
5. A NUMERIC feature consists of:
 - a. A name which may contain any print character other than ', ', '=' , and '# '
 - b. The '# ' character
 - c. A positive integer which conforms to the C-language convention for integer constants
6. A STRING feature consists of:
 - a. A name which may contain any print character other than ', ', '=' , and '# '

2. A BOOLEAN feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

3. A NUMERIC feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

4. A STRING feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

5. Graph characters are those characters for which *isgraph()* returns non-zero.

6. Print characters are those characters for which *isprint()* returns non-zero.

- b. The '=' character
 - c. A string which may contain any print characters other than ', '
7. White space immediately following a ', ' is ignored.
 8. Comments are lines consisting of zero or more whitespace characters followed by a '#' sign, followed by zero or more non-<newline> characters and terminated by a <newline>.
 9. A header line must begin in column one.
 10. A feature line must not begin in column one.
 11. Blank lines are ignored.

7.1.3 Defined Capabilities

The Open Group defines the capabilities listed in the following table. All X/Open-compliant implementations must accept each of these capabilities in an entry in a **terminfo** source file. Implementations use this information to determine how properly to operate the current terminal. In addition, implementations return any of the current terminal's capabilities when the application calls the query functions listed in *tigetflag()* (on page 232).

The table of capabilities has the following columns:

| | |
|--------------------|--|
| Variable | Names for use by the Curses functions that operate on the terminfo database. These names are reserved and the application must not define them. |
| Capname | The short name for a capability specified in the terminfo source file. It is used for updating the source file and by the <i>tput</i> command. |
| Description | A description of the capability. In some cases a notation "#1", "#2", etc. is used to refer to parameters for an associated call to <i>tiparm()</i> . |

Booleans

| Variable | Capname | Description |
|---------------------------------|--------------|---|
| <i>auto_left_margin</i> | bw | cub1 wraps from column 0 to last column |
| <i>auto_right_margin</i> | am | Terminal has automatic margins |
| <i>back_color_erase</i> | bce | Screen erased with background color |
| <i>can_change</i> | ccc | Terminal can redefine existing color |
| <i>ceol_standout_glitch</i> | xhp | Standout not erased by overwriting (hp) |
| <i>col_addr_glitch</i> | xhpa | Only positive motion for hpa/mhpa caps |
| <i>cpi_changes_res</i> | cpix | Changing character pitch changes resolution |
| <i>cr_cancels_micro_mode</i> | crxm | Using cr turns off micro mode |
| <i>dest_tabs_magic_smso</i> | xt | Destructive tabs, magic smso char (t1061) |
| <i>eat_newline_glitch</i> | xenl | Newline ignored after 80 columns (Concept) |
| <i>erase_overstrike</i> | eo | Can erase overstrikes with a <blank> |
| <i>generic_type</i> | gn | Generic line type (e.g., dialup, switch) |
| <i>hard_copy</i> | hc | Hardcopy terminal |
| <i>hard_cursor</i> | chts | Cursor is hard to see |
| <i>has_meta_key</i> | km | Has a meta key (shift, sets parity bit) |
| <i>has_print_wheel</i> | daisy | Printer needs operator to change character set |
| <i>has_status_line</i> | hs | Has extra "status line" |
| <i>hue_lightness_saturation</i> | hls | Terminal uses only HLS color notation (Tektronix) |
| <i>insert_null_glitch</i> | in | Insert mode distinguishes nulls |

| Variable | Capname | Description |
|-------------------------------|--------------|---|
| <i>lpi_changes_res</i> | lpix | Changing line pitch changes resolution |
| <i>memory_above</i> | da | Display may be retained above the screen |
| <i>memory_below</i> | db | Display may be retained below the screen |
| <i>move_insert_mode</i> | mir | Safe to move while in insert mode |
| <i>move_standout_mode</i> | msgr | Safe to move in standout modes |
| <i>needs_xon_xoff</i> | nxon | Padding won't work, xon/xoff required |
| <i>no_esc_ctlc</i> | xsb | Beehive (f1=escape, f2=ctrl C) |
| <i>no_pad_char</i> | npc | Pad character doesn't exist |
| <i>non_dest_scroll_region</i> | ndscr | Scrolling region is non-destructive |
| <i>non_rev_rmcup</i> | nrrmc | smcup does not reverse rncup |
| <i>over_strike</i> | os | Terminal overstrikes on hard-copy terminal |
| <i>prtr_silent</i> | mc5i | Printer won't echo on screen |
| <i>row_addr_glitch</i> | xvpa | Only positive motion for vpa/mvpa caps |
| <i>semi_auto_right_margin</i> | sam | Printing in last column causes cr |
| <i>status_line_esc_ok</i> | eslok | Escape can be used on the status line |
| <i>tilde_glitch</i> | hz | Hazeltine; can't print tilde (~) |
| <i>transparent_underline</i> | ul | Underline character overstrikes |
| <i>xon_xoff</i> | xon | Terminal uses xon/xoff handshaking |

Numbers

| Variable | Capname | Description |
|-----------------------------|---------------|---|
| <i>bit_image_entwining</i> | bitwin | Number of passes for each bit-map row |
| <i>bit_image_type</i> | bitype | Type of bit image device |
| <i>buffer_capacity</i> | bufsz | Number of bytes buffered before printing |
| <i>buttons</i> | btns | Number of buttons on the mouse |
| <i>columns</i> | cols | Number of columns in a line |
| <i>dot_horz_spacing</i> | spinh | Spacing of dots horizontally in dots per inch |
| <i>dot_vert_spacing</i> | spinv | Spacing of pins vertically in pins per inch |
| <i>init_tabs</i> | it | Initial number of columns between tab positions |
| <i>label_height</i> | lh | Number of rows in each label |
| <i>label_width</i> | lw | Number of columns in each label |
| <i>lines</i> | lines | Number of lines on a screen or a page |
| <i>lines_of_memory</i> | lm | Lines of memory if > lines ; 0 means varies |
| <i>max_attributes</i> | ma | Maximum combined video attributes terminal can display |
| <i>magic_cookie_glitch</i> | xmc | Number of <blank> characters left by smso or rmso |
| <i>max_colors</i> | colors | Maximum number of colors on the screen |
| <i>max_micro_address</i> | maddr | Maximum value in micro_..._address |
| <i>max_micro_jump</i> | mjump | Maximum value in parm_..._micro |
| <i>max_pairs</i> | pairs | Maximum number of color-pairs on the screen |
| <i>maximum_windows</i> | wnum | Maximum number of definable windows |
| <i>micro_col_size</i> | mcs | Character step size when in micro mode |
| <i>micro_line_size</i> | mls | Line step size when in micro mode |
| <i>no_color_video</i> | ncv | Video attributes that can't be used with colors |
| <i>num_labels</i> | nlab | Number of labels on screen (start at 1) |
| <i>number_of_pins</i> | npins | Number of pins in print-head |
| <i>output_res_char</i> | orc | Horizontal resolution in units per character |
| <i>output_res_line</i> | orl | Vertical resolution in units per line |
| <i>output_res_horz_inch</i> | orhi | Horizontal resolution in units per inch |

| Variable | Capname | Description |
|-----------------------------|--------------|--|
| <i>output_res_vert_inch</i> | orvi | Vertical resolution in units per inch |
| <i>padding_baud_rate</i> | pb | Lowest baud rate where padding needed |
| <i>print_rate</i> | cps | Print rate in characters per second |
| <i>virtual_terminal</i> | vt | Virtual terminal number |
| <i>wide_char_size</i> | widcs | Character step size when in double-wide mode |
| <i>width_status_line</i> | wsl | Number of columns in status line |

Strings

| Variable | Capname | Description |
|----------------------------------|---------------|--|
| <i>acs_chars</i> | acsc | Graphic charset pairs aAbBcC |
| <i>alt_scancode_esc</i> | scesa | Alternate escape for scancode emulation (default is for VT100) |
| <i>back_tab</i> | cbt | Back tab |
| <i>bell</i> | bel | Audible signal (bell) |
| <i>bit_image_carriage_return</i> | bicr | Move to beginning of same row |
| <i>bit_image_newline</i> | binel | Move to next row of the bit image |
| <i>bit_image_repeat</i> | birep | Repeat bit-image cell #1 #2 times |
| <i>carriage_return</i> | cr | Carriage-return |
| <i>change_char_pitch</i> | cpi | Change number of characters per inch |
| <i>change_line_pitch</i> | lpi | Change number of lines per inch |
| <i>change_res_horz</i> | chr | Change horizontal resolution |
| <i>change_res_vert</i> | cvr | Change vertical resolution |
| <i>change_scroll_region</i> | csr | Change to lines #1 through #2 (VT100) |
| <i>char_padding</i> | rmp | Like ip but when in replace mode |
| <i>char_set_names</i> | csnm | Returns a list of character set names |
| <i>clear_all_tabs</i> | tbc | Clear all tab stops |
| <i>clear_margins</i> | mgc | Clear all margins (top, bottom, and sides) |
| <i>clear_screen</i> | clear | Clear screen and home cursor |
| <i>clr_bol</i> | el1 | Clear to beginning of line, inclusive |
| <i>clr_eol</i> | el | Clear to end of line |
| <i>clr_eos</i> | ed | Clear to end of display |
| <i>code_set_init</i> | csin | Init sequence for multiple codesets |
| <i>color_names</i> | colorm | Give name for color #1 |
| <i>column_address</i> | hpa | Set horizontal position to absolute #1 |
| <i>command_character</i> | cmdch | Terminal settable cmd character in prototype |
| <i>create_window</i> | cwin | Define win #1 to go from #2,#3 to #4,#5 |
| <i>cursor_address</i> | cup | Move to row #1 col #2 |
| <i>cursor_down</i> | cud1 | Down one line |
| <i>cursor_home</i> | home | Home cursor (if no cup) |
| <i>cursor_invisible</i> | civis | Make cursor invisible |
| <i>cursor_left</i> | cub1 | Move left one space. |
| <i>cursor_mem_address</i> | mrcup | Memory-relative cursor addressing |
| <i>cursor_normal</i> | cnorm | Make cursor appear normal (undo cvvis/civis) |
| <i>cursor_right</i> | cuf1 | Non-destructive space (cursor or carriage right) |
| <i>cursor_to_ll</i> | ll | Last line, first column (if no cup) |
| <i>cursor_up</i> | cuu1 | Upline (cursor up) |
| <i>cursor_visible</i> | cvvis | Make cursor very visible |
| <i>define_bit_image_region</i> | defbi | Define rectangular bit-image region |
| <i>define_char</i> | defc | Define a character in a character set |

| Variable | Capname | Description |
|----------------------------------|---------------|---|
| <i>delete_character</i> | dch1 | Delete character |
| <i>delete_line</i> | dl1 | Delete line |
| <i>device_type</i> | devt | Indicate language/codeset support |
| <i>dial_phone</i> | dial | Dial phone number #1 |
| <i>dis_status_line</i> | dsl | Disable status line |
| <i>display_clock</i> | dclk | Display time-of-day clock |
| <i>display_pc_char</i> | dispc | Display PC character |
| <i>down_half_line</i> | hd | Half-line down (forward 1/2 linefeed) |
| <i>ena_acs</i> | enacs | Enable alternate character set |
| <i>end_bit_image_region</i> | endbi | End a bit-image region |
| <i>enter_alt_charset_mode</i> | smacs | Start alternate character set |
| <i>enter_am_mode</i> | smam | Turn on automatic margins |
| <i>enter_blink_mode</i> | blink | Turn on blinking |
| <i>enter_bold_mode</i> | bold | Turn on bold (extra bright) mode |
| <i>enter_ca_mode</i> | smcup | String to begin programs that use cup |
| <i>enter_delete_mode</i> | smdc | Delete mode (enter) |
| <i>enter_dim_mode</i> | dim | Turn on half-bright mode |
| <i>enter_doublewide_mode</i> | swidm | Enable double wide printing |
| <i>enter_draft_quality</i> | sdrfq | Set draft quality print |
| <i>enter_horizontal_hl_mode</i> | ehhlm | Turn on horizontal highlight mode |
| <i>enter_insert_mode</i> | smir | Insert mode (enter) |
| <i>enter_italics_mode</i> | sitm | Enable italics |
| <i>enter_left_hl_mode</i> | elhlm | Turn on left highlight mode |
| <i>enter_leftward_mode</i> | slm | Enable leftward carriage motion |
| <i>enter_low_hl_mode</i> | elohlm | Turn on low highlight mode |
| <i>enter_micro_mode</i> | smicm | Enable micro motion capabilities |
| <i>enter_near_letter_quality</i> | snlq | Set near-letter quality print |
| <i>enter_normal_quality</i> | snrmq | Set normal quality print |
| <i>enter_pc_charset_mode</i> | smpch | Enter PC character display mode |
| <i>enter_protected_mode</i> | prot | Turn on protected mode |
| <i>enter_reverse_mode</i> | rev | Turn on reverse video mode |
| <i>enter_right_hl_mode</i> | erhlm | Turn on right highlight mode |
| <i>enter_scancode_mode</i> | smsc | Enter PC scancode mode |
| <i>enter_secure_mode</i> | invis | Turn on blank mode (characters invisible) |
| <i>enter_shadow_mode</i> | sshm | Enable shadow printing |
| <i>enter_standout_mode</i> | smso | Begin standout mode |
| <i>enter_subscript_mode</i> | ssubm | Enable subscript printing |
| <i>enter_superscript_mode</i> | ssupm | Enable superscript printing |
| <i>enter_top_hl_mode</i> | ethlm | Turn on top highlight mode |
| <i>enter_underline_mode</i> | smul | Start underscore mode |
| <i>enter_upward_mode</i> | sum | Enable upward carriage motion |
| <i>enter_vertical_hl_mode</i> | evhlm | Turn on vertical highlight mode |
| <i>enter_xon_mode</i> | smxon | Turn on xon/xoff handshaking |
| <i>erase_chars</i> | ech | Erase #1 characters |
| <i>exit_alt_charset_mode</i> | rmacs | End alternate character set |
| <i>exit_am_mode</i> | rmam | Turn off automatic margins |
| <i>exit_attribute_mode</i> | sgr0 | Turn off all attributes |
| <i>exit_ca_mode</i> | rmcup | String to end programs that use cup |
| <i>exit_delete_mode</i> | rmdc | End delete mode |
| <i>exit_doublewide_mode</i> | rwidm | Disable double wide printing |
| <i>exit_insert_mode</i> | rmir | End insert mode |

| Variable | Capname | Description |
|------------------------------|--------------|---|
| <i>exit_italics_mode</i> | ritm | Disable italics |
| <i>exit_leftward_mode</i> | rlm | Enable rightward (normal) carriage motion |
| <i>exit_micro_mode</i> | rmicm | Disable micro motion capabilities |
| <i>exit_pc_charset_mode</i> | rmpch | Disable PC character display mode |
| <i>exit_scancode_mode</i> | rmsc | Disable PC scancode mode |
| <i>exit_shadow_mode</i> | rshm | Disable shadow printing |
| <i>exit_standout_mode</i> | rmso | End standout mode |
| <i>exit_subscript_mode</i> | rsubm | Disable subscript printing |
| <i>exit_superscript_mode</i> | rsupm | Disable superscript printing |
| <i>exit_underline_mode</i> | rmul | End underscore mode |
| <i>exit_upward_mode</i> | rum | Enable downward (normal) carriage motion |
| <i>exit_xon_mode</i> | rmxon | Turn off xon/xoff handshaking |
| <i>fixed_pause</i> | pause | Pause for 2-3 seconds |
| <i>flash_hook</i> | hook | Flash the switch hook |
| <i>flash_screen</i> | flash | Visible bell (may move cursor) |
| <i>form_feed</i> | ff | Hardcopy terminal page eject |
| <i>from_status_line</i> | fsl | Return from status line |
| <i>get_mouse</i> | getm | Curses should get button events |
| <i>goto_window</i> | wingo | Go to window #1 |
| <i>hangup</i> | hup | Hang-up phone |
| <i>init_1string</i> | is1 | Terminal or printer initialization string |
| <i>init_2string</i> | is2 | Terminal or printer initialization string |
| <i>init_3string</i> | is3 | Terminal or printer initialization string |
| <i>init_file</i> | if | Name of initialization file |
| <i>init_prog</i> | ipro | Path name of program for initialization |
| <i>initialize_color</i> | initc | Set color #1 to RGB #2, #3, #4 |
| <i>initialize_pair</i> | initp | Set color-pair #1 to fg #2, bg #3 |
| <i>insert_character</i> | ich1 | Insert character |
| <i>insert_line</i> | il1 | Add new blank line |
| <i>insert_padding</i> | ip | Insert pad after character inserted |

The `key_` strings are sent by specific keys. The `key_` descriptions include the macro, defined in `< curses.h >`, for the code returned by `getch()` when the key is pressed (see `getch()`).

| Variable | Capname | Description |
|----------------------|--------------|-----------------------------------|
| <i>key_a1</i> | ka1 | Upper left of keypad |
| <i>key_a3</i> | ka3 | Upper right of keypad |
| <i>key_b2</i> | kb2 | Center of keypad |
| <i>key_backspace</i> | kbs | Sent by backspace key |
| <i>key_beg</i> | kbeg | Sent by beg(inning) key |
| <i>key_btab</i> | kcbt | Sent by back-tab key |
| <i>key_c1</i> | kc1 | Lower left of keypad |
| <i>key_c3</i> | kc3 | Lower right of keypad |
| <i>key_cancel</i> | kcan | Sent by cancel key |
| <i>key_catab</i> | ktbc | Sent by clear-all-tabs key |
| <i>key_clear</i> | kclr | Sent by clear-screen or erase key |
| <i>key_close</i> | kclo | Sent by close key |
| <i>key_command</i> | kcmd | Sent by cmd (command) key |
| <i>key_copy</i> | kcpy | Sent by copy key |
| <i>key_create</i> | kcrt | Sent by create key |
| <i>key_ctab</i> | kctab | Sent by clear-tab key |

| Variable | Capname | Description |
|----------------------|--------------|---|
| <i>key_dc</i> | kdch1 | Sent by delete-character key |
| <i>key_dl</i> | kd11 | Sent by delete-line key |
| <i>key_down</i> | kcud1 | Sent by terminal down-arrow key |
| <i>key_eic</i> | krmir | Sent by rmir or smir in insert mode |
| <i>key_end</i> | kend | Sent by end key |
| <i>key_enter</i> | kent | Sent by enter/send key |
| <i>key_eol</i> | kel | Sent by clear-to-end-of-line key |
| <i>key_eos</i> | ked | Sent by clear-to-end-of-screen key |
| <i>key_exit</i> | kext | Sent by exit key |
| <i>key_f0</i> | kf0 | Sent by function key f0 |
| <i>key_f1</i> | kf1 | Sent by function key f1 |
| . | . | . |
| . | . | similarly for f2 through f61 |
| . | . | . |
| <i>key_f62</i> | kf62 | Sent by function key f62 |
| <i>key_f63</i> | kf63 | Sent by function key f63 |
| <i>key_find</i> | kfnd | Sent by find key |
| <i>key_help</i> | khlp | Sent by help key |
| <i>key_home</i> | khome | Sent by home key |
| <i>key_ic</i> | kich1 | Sent by ins-char/enter ins-mode key |
| <i>key_il</i> | kil1 | Sent by insert-line key |
| <i>key_left</i> | kcub1 | Sent by terminal left-arrow key |
| <i>key_ll</i> | kll | Sent by home-down key |
| <i>key_mark</i> | kmrk | Sent by mark key |
| <i>key_message</i> | kmsg | Sent by message key |
| <i>key_mouse</i> | kmous | 0631, Mouse event has occurred |
| <i>key_move</i> | kmov | Sent by move key |
| <i>key_next</i> | knxt | Sent by next-object key |
| <i>key_npage</i> | knp | Sent by next-page key |
| <i>key_open</i> | kopn | Sent by open key |
| <i>key_options</i> | kopt | Sent by options key |
| <i>key_ppage</i> | kpp | Sent by previous-page key |
| <i>key_previous</i> | kprv | Sent by previous-object key |
| <i>key_print</i> | kpri | Sent by print or copy key |
| <i>key_redo</i> | krdo | Sent by redo key |
| <i>key_reference</i> | krf | Sent by ref(erence) key |
| <i>key_refresh</i> | krfr | Sent by refresh key |
| <i>key_replace</i> | krpl | Sent by replace key |
| <i>key_restart</i> | krst | Sent by restart key |
| <i>key_resume</i> | kres | Sent by resume key |
| <i>key_right</i> | kcuf1 | Sent by terminal right-arrow key |
| <i>key_save</i> | ksav | Sent by save key |
| <i>key_sbeg</i> | kBEG | Sent by shifted beginning key |
| <i>key_scancel</i> | kCAN | Sent by shifted cancel key |
| <i>key_scommand</i> | kCMD | Sent by shifted command key |
| <i>key_scopy</i> | kCPY | Sent by shifted copy key |
| <i>key_screate</i> | kCRT | Sent by shifted create key |
| <i>key_sdc</i> | kDC | Sent by shifted delete-char key |
| <i>key_sdl</i> | kDL | Sent by shifted delete-line key |
| <i>key_select</i> | kslt | Sent by select key |
| <i>key_send</i> | kEND | Sent by shifted end key |

| Variable | Capname | Description |
|-----------------------------|--------------|---|
| <i>key_seol</i> | kEOL | Sent by shifted clear-line key |
| <i>key_sexit</i> | kEXT | Sent by shifted exit key |
| <i>key_sf</i> | kind | Sent by scroll-forward/down key |
| <i>key_sfind</i> | kFND | Sent by shifted find key |
| <i>key_shelp</i> | kHLP | Sent by shifted help key |
| <i>key_shome</i> | kHOM | Sent by shifted home key |
| <i>key_sic</i> | kIC | Sent by shifted input key |
| <i>key_sleft</i> | kLFT | Sent by shifted left-arrow key |
| <i>key_smessage</i> | kMSG | Sent by shifted message key |
| <i>key_smove</i> | kMOV | Sent by shifted move key |
| <i>key_snext</i> | kNXT | Sent by shifted next key |
| <i>key_soptions</i> | kOPT | Sent by shifted options key |
| <i>key_sprevious</i> | kPRV | Sent by shifted prev key |
| <i>key_sprint</i> | kPRT | Sent by shifted print key |
| <i>key_sr</i> | kri | Sent by scroll-backward/up key |
| <i>key_sredo</i> | krDO | Sent by shifted redo key |
| <i>key_sreplace</i> | krPL | Sent by shifted replace key |
| <i>key_sright</i> | krIT | Sent by shifted right-arrow key |
| <i>key_sresume</i> | kRES | Sent by shifted resume key |
| <i>key_ssave</i> | kSAV | Sent by shifted save key |
| <i>key_ssuspend</i> | kSPD | Sent by shifted suspend key |
| <i>key_stab</i> | khts | Sent by set-tab key |
| <i>key_sundo</i> | kUND | Sent by shifted undo key |
| <i>key_suspend</i> | kspd | Sent by suspend key |
| <i>key_undo</i> | kund | Sent by undo key |
| <i>key_up</i> | kcuu1 | Sent by terminal up-arrow key |
| <i>keypad_local</i> | rmkx | Out of "keypad-transmit" mode |
| <i>keypad_xmit</i> | smkx | Put terminal in "keypad-transmit" mode |
| <i>lab_f0</i> | lf0 | Labels on function key f0 if not f0 |
| <i>lab_f1</i> | lf1 | Labels on function key f1 if not f1 |
| <i>lab_f2</i> | lf2 | Labels on function key f2 if not f2 |
| <i>lab_f3</i> | lf3 | Labels on function key f3 if not f3 |
| <i>lab_f4</i> | lf4 | Labels on function key f4 if not f4 |
| <i>lab_f5</i> | lf5 | Labels on function key f5 if not f5 |
| <i>lab_f6</i> | lf6 | Labels on function key f6 if not f6 |
| <i>lab_f7</i> | lf7 | Labels on function key f7 if not f7 |
| <i>lab_f8</i> | lf8 | Labels on function key f8 if not f8 |
| <i>lab_f9</i> | lf9 | Labels on function key f9 if not f9 |
| <i>lab_f10</i> | lf10 | Labels on function key f10 if not f10 |
| <i>label_format</i> | fln | Label format |
| <i>label_off</i> | rmln | Turn off soft labels |
| <i>label_on</i> | smln | Turn on soft labels |
| <i>meta_off</i> | rmm | Turn off "meta mode" |
| <i>meta_on</i> | smm | Turn on "meta mode" (8th bit) |
| <i>micro_column_address</i> | mhpa | Like column_address for micro adjustment |
| <i>micro_down</i> | mcud1 | Like cursor_down for micro adjustment |
| <i>micro_left</i> | mcub1 | Like cursor_left for micro adjustment |
| <i>micro_right</i> | mcuf1 | Like cursor_right for micro adjustment |
| <i>micro_row_address</i> | mvpa | Like row_address for micro adjustment |
| <i>micro_up</i> | mcuu1 | Like cursor_up for micro adjustment |
| <i>mouse_info</i> | minfo | Mouse status information |

| Variable | Capname | Description |
|--------------------------|---------------|---|
| <i>newline</i> | nel | Newline (behaves like cr followed by lf) |
| <i>order_of_pins</i> | porder | Matches software bits to print-head pins |
| <i>orig_colors</i> | oc | Set all color(-pair)s to the original ones |
| <i>orig_pair</i> | op | Set default color-pair to the original one |
| <i>pad_char</i> | pad | Pad character (rather than null) |
| <i>parm_dch</i> | dch | Delete #1 chars |
| <i>parm_delete_line</i> | dl | Delete #1 lines |
| <i>parm_down_cursor</i> | cud | Move down #1 lines. |
| <i>parm_down_micro</i> | mcud | Like parm_down_cursor for micro adjust. |
| <i>parm_ich</i> | ich | Insert #1 <blank> chars |
| <i>parm_index</i> | indn | Scroll forward #1 lines. |
| <i>parm_insert_line</i> | il | Add #1 new blank lines |
| <i>parm_left_cursor</i> | cub | Move cursor left #1 spaces |
| <i>parm_left_micro</i> | mcub | Like parm_left_cursor for micro adjust. |
| <i>parm_right_cursor</i> | cuf | Move right #1 spaces. |
| <i>parm_right_micro</i> | mcuf | Like parm_right_cursor for micro adjust. |
| <i>parm_rindex</i> | rin | Scroll backward #1 lines. |
| <i>parm_up_cursor</i> | cuu | Move cursor up #1 lines. |
| <i>parm_up_micro</i> | mcuu | Like parm_up_cursor for micro adjust. |
| <i>pc_term_options</i> | pctrm | PC terminal options |
| <i>pkey_key</i> | pfkey | Prog funct key #1 to type string #2 |
| <i>pkey_local</i> | pfloc | Prog funct key #1 to execute string #2 |
| <i>pkey_plab</i> | pfxl | Prog key #1 to xmit string #2 and show string #3 |
| <i>pkey_xmit</i> | pfx | Prog funct key #1 to xmit string #2 |
| <i>plab_norm</i> | pln | Prog label #1 to show string #2 |
| <i>print_screen</i> | mc0 | Print contents of the screen |
| <i>prtr_non</i> | mc5p | Turn on the printer for #1 bytes |
| <i>prtr_off</i> | mc4 | Turn off the printer |
| <i>prtr_on</i> | mc5 | Turn on the printer |
| <i>pulse</i> | pulse | Select pulse dialing |
| <i>quick_dial</i> | qdial | Dial phone number #1, without progress detection |
| <i>remove_clock</i> | rmclk | Remove time-of-day clock |
| <i>repeat_char</i> | rep | Repeat char #1 #2 times |
| <i>req_for_input</i> | rfi | Send next input char (for ptys) |
| <i>req_mouse_pos</i> | reqmp | Request mouse position report |
| <i>reset_1string</i> | rs1 | Reset terminal completely to sane modes |
| <i>reset_2string</i> | rs2 | Reset terminal completely to sane modes |
| <i>reset_3string</i> | rs3 | Reset terminal completely to sane modes |
| <i>reset_file</i> | rf | Name of file containing reset string |
| <i>restore_cursor</i> | rc | Restore cursor to position of last sc |
| <i>row_address</i> | vpa | Set vertical position to absolute #1 |
| <i>save_cursor</i> | sc | Save cursor position |
| <i>scancode_escape</i> | scesc | Escape for scancode emulation |
| <i>scroll_forward</i> | ind | Scroll text up |
| <i>scroll_reverse</i> | ri | Scroll text down |
| <i>select_char_set</i> | scs | Select character set |
| <i>set0_des_seq</i> | s0ds | Shift into codeset 0 (EUC set 0, ASCII) |
| <i>set1_des_seq</i> | s1ds | Shift into codeset 1 |
| <i>set2_des_seq</i> | s2ds | Shift into codeset 2 |
| <i>set3_des_seq</i> | s3ds | Shift into codeset 3 |
| <i>set_a_attributes</i> | sgr1 | Define second set of video attributes #1-#6 |

| Variable | Capname | Description |
|-------------------------------|-----------------|--|
| <i>set_a_background</i> | setab | Set background color to #1 using ANSI escape |
| <i>set_a_foreground</i> | setaf | Set foreground color to #1 using ANSI escape |
| <i>set_attributes</i> | sgr | Define first set of video attributes #1-#9 |
| <i>set_background</i> | setb | Set background color to #1 |
| <i>set_bottom_margin</i> | smgb | Set bottom margin at current line |
| <i>set_bottom_margin_parm</i> | smgbp | Set bottom margin at line #1 or #2 lines from bottom |
| <i>set_clock</i> | sclk | Set clock to hours (#1), minutes (#2), seconds (#3) |
| <i>set_color_band</i> | setcolor | Change to ribbon color #1 |
| <i>set_color_pair</i> | scp | Set current color pair to #1 |
| <i>set_foreground</i> | setf | Set foreground color to #1 |
| <i>set_left_margin</i> | smgl | Set left margin at current column |
| <i>set_left_margin_parm</i> | smglp | Set left (right) margin at column #1 (#2) |
| <i>set_lr_margin</i> | smglr | Sets both left and right margins |
| <i>set_page_length</i> | slines | Set page length to #1 lines |
| <i>set_pglen_inch</i> | slength | Set page length to #1 hundredth of an inch |
| <i>set_right_margin</i> | smgr | Set right margin at current column |
| <i>set_right_margin_parm</i> | smgrp | Set right margin at column #1 |
| <i>set_tab</i> | hts | Set a tab in all rows, current column |
| <i>set_tb_margin</i> | smgtb | Sets both top and bottom margins |
| <i>set_top_margin</i> | smgt | Set top margin at current line |
| <i>set_top_margin_parm</i> | smgtp | Set top (bottom) margin at line #1 (#2) |
| <i>set_window</i> | wind | Current window is lines #1-#2 cols #3-#4 |
| <i>start_bit_image</i> | sbim | Start printing bit image graphics |
| <i>start_char_set_def</i> | scsd | Start definition of a character set |
| <i>stop_bit_image</i> | rbim | End printing bit image graphics |
| <i>stop_char_set_def</i> | rcsd | End definition of a character set |
| <i>subscript_characters</i> | subcs | List of "subscript-able" characters |
| <i>superscript_characters</i> | supcs | List of "superscript-able" characters |
| <i>tab</i> | ht | Tab to next 8-space hardware tab stop |
| <i>these_cause_cr</i> | docr | Printing any of these chars causes cr |
| <i>to_status_line</i> | tsl | Go to status line, col #1 |
| <i>tone</i> | tone | Select touch tone dialing |
| <i>user0</i> | u0 | User string 0 |
| <i>user1</i> | u1 | User string 1 |
| <i>user2</i> | u2 | User string 2 |
| <i>user3</i> | u3 | User string 3 |
| <i>user4</i> | u4 | User string 4 |
| <i>user5</i> | u5 | User string 5 |
| <i>user6</i> | u6 | User string 6 |
| <i>user7</i> | u7 | User string 7 |
| <i>user8</i> | u8 | User string 8 |
| <i>user9</i> | u9 | User string 9 |
| <i>underline_char</i> | uc | Underscore one char and move past it |
| <i>up_half_line</i> | hu | Half-line up (reverse 1/2 linefeed) |
| <i>wait_tone</i> | wait | Wait for dial tone |
| <i>xoff_character</i> | xoffc | X-off character |
| <i>xon_character</i> | xonc | X-on character |
| <i>zero_motion</i> | zerom | No motion for the subsequent character |

7.1.4 Sample Entry

The following entry describes the AT&T 610 terminal:

```
610 | 610bct | ATT610 | att610 | AT&T 610; 80 column; 98key keyboard,
    am, eslok, hs, mir, msgr, xenl, xon,
    cols#80, it#8, lh#2, lines#24, lw#8, nlab#8, wsl#80,
    acsc=`\aaffggjjkkllmmnnooppqrrssttuuvvwwxxyyzz{|}|}~`,
    bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z,
    civis=\E[?25l, clear=\E[H\E[J, cnorm=\E[?25h\E[?12l,
    cr=\r, csr=\E[%i%p1%d;%p2%dr, cub=\E[%p1%dD, cubl=\b,
    cud=\E[%p1%dB, cudl=\E[B, cuf=\E[%p1%DC, cuf1=\E[C,
    cup=\E[%i%p1%d;%p2%dH, cuu=\E[%p1%DA, cuul=\E[A,
    cvvis=\E[?12;25h, dch=\E[%p1%DP, dch1=\E[P, dim=\E[2m,
    dl=\E[%p1%DM, dll=\E[M, ed=\E[J, el=\E[K, ell=\E[1K,
    flash=\E[?5h$<200>\E[?5l, fsl=\E8, home=\E[H, ht=\t,
    ich=\E[%p1%D@, il=\E[%p1%DL, ill=\E[L, ind=\ED, .ind=\ED$<9>,
    invis=\E[8m,
    is1=\E[8;0 | \E[?3;4;5;13;15l\E[13;20l\E[?7h\E[12h\E(B\E)0,
    is2=\E[0m^O, is3=\E(B\E)0, kLFT=\E[\s@, kRIT=\E[\sA,
    kbs=^H, kcbt=\E[Z, kclr=\E[2J, kcub1=\E[D, kcud1=\E[B,
    kcufl1=\E[C, kcuul1=\E[A, kfp=\EOc, kfp0=\ENp,
    kfp1=\ENq, kfp2=\ENr, kfp3=\ENs, kfp4=\ENT, kfi=\Eod,
    kfb=\EOe, kf4=\EOf, kf(CW=\EOg, kf6=\EOh, kf7=\EOi,
    kf8=\EOj, kf9=\ENo, khome=\E[H, kind=\E[S, kri=\E[T,
    ll=\E[24H, mc4=\E[?4i, mc5=\E[?5i, nel=\EE,
    pfx1=\E[%p1%d;%p2%l%02dq%?%p1%{9}%<?%t\s\s\sF%p1%ld\s\s\s\s\s
\s\s\s\s\s\s%:%p2%s,
    pln=\E[%p1%d;0;0;0q%p2%:-16.16s, rc=\E8, rev=\E[7m,
    ri=\EM, rmacs=^O, rmir=\E[4l, rmln=\E[2p, rmso=\E[m,
    rmul=\E[m, rs2=\Ec\E[?3l, sc=\E7,
    sgr=\E[0%?%p6%t;1%:%?%p5%t;2%:%?%p2%t;4%:%?%p4%t;5%:
%?%p3%p1% | %t;7%:%?%p7%t;8%;m%?%p9%t^N^e^O%;,
    sgr0=\E[m^O, smacs=^N, smir=\E[4h, smln=\E[p,
    smso=\E[7m, smul=\E[4m, tsl=\E7\E[25;%i%p1%dx,
```

7.1.5 Types of Capabilities in the Sample Entry

The sample entry shows the formats for the three types of **terminfo** capabilities: Boolean, numeric, and string. All capabilities specified in the **terminfo** source file must be followed by commas, including the last capability in the source file. In **terminfo** source files, capabilities are referenced by their capability names (as shown in the **Capname** column of the previous tables).

Boolean Capabilities

A boolean capability is true if its **Capname** is present in the entry, and false if its **Capname** is not present in the entry.

The '@' character following a **Capname** is used to explicitly declare that a boolean capability is false, in situations described in [Section A.1.16](#) (on page 366).

Numeric Capabilities

Numeric capabilities are followed by the character '#' and then a positive integer value. The example assigns the value 80 to the **cols** numeric capability by coding:

```
cols#80
```

Values for numeric capabilities may be specified in decimal, octal, or hexadecimal, using normal C-language conventions.

String Capabilities

String-valued capabilities such as **el** (clear to end of line sequence) are listed by the **Capname**, an '=', and a string ended by the next occurrence of a comma.

A delay in milliseconds may appear anywhere in such a capability, preceded by '\$' and enclosed in angle brackets, as in **el=\EK\$<3>**. The Curses implementation achieves delays by outputting to the terminal an appropriate number of system-defined padding characters. The *tputs()* function provides delays when used to send such a capability to the terminal.

The delay can be any of the following: a number, a number followed by an asterisk, such as **5***, a number followed by a slash, such as **5/**, or a number followed by both, such as **5*/**.

- A '*' shows that the required delay is proportional to the number of lines affected by the operation, and the amount given is the delay required per affected unit. (In the case of insert characters, the factor is still the number of lines affected. This is always 1 unless the device has **in** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form **3.5** to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)
- A '/' indicates that the delay is mandatory and padding characters are transmitted regardless of the setting of **xon**. If '/' is not specified or if a device has **xon** defined, the delay information is advisory and is only used for cost estimates or when the device is in raw mode. However, any delay specified for **bel** or **flash** is treated as mandatory.

The following notation is valid in **terminfo** source files for specifying special characters:

| Notation | Represents Character |
|------------------------|---|
| $\^x$ | Control- <i>x</i> (for any appropriate <i>x</i>) |
| \a | Alert |
| \b | Backspace |
| \E or \e | An ESCAPE character |
| \f | Form feed |
| \l | Linefeed |
| \n | Newline |
| \r | Carriage-return |
| \s | Space |
| \t | Tab |
| \^ | Caret (^) |
| \ | Backslash (\) |
| \, | Comma (,) |
| \: | Colon (:) |
| \0 | Null |
| \nnn | Any character, specified as three octal digits |

(See the **XBD** specification, General Terminal Interface.)

Commented-out Capabilities

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example in [Section 7.1.4](#) (on page 349). Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

A.1 Device Capabilities

A.1.1 Basic Capabilities

The number of columns on each line for the device is given by the **cols** numeric capability. If the device has a screen, then the number of lines on the screen is given by the **lines** capability. If the device wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the device is a printing terminal, with no soft copy unit, specify both **hc** and **os**. If there is a way to move the cursor to the left edge of the current row, specify this as **cr**. (Normally this will be <carriage-return>, <control-M>.) If there is a way to produce an audible signal (such as a bell or a beep), specify it as **bel**. If, like most devices, the device uses the xon/xoff flow-control protocol, specify **xon**.

If there is a way to move the cursor one position to the left (such as backspace), that capability should be given as **cub1**. Similarly, sequences to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**, respectively. These local cursor motions must not alter the text they pass over; for example, you would not normally use **cuf1** =\s because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in **terminfo** are undefined at the left and top edges of a screen terminal. Programs should never attempt to backspace around the left edge, unless **bw** is specified, and should never attempt to go up locally off the top. To scroll text up, a program goes to the bottom left corner of the screen and sends the **ind** (index) string. To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin**. These versions have the same semantics as **ind** and **ri**, except that they take one argument and scroll the number of lines specified by that argument. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. Backward motion from the left edge of the screen is possible only when **bw** is specified. In this case, **cub1** will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the device has switch-selectable automatic margins, **am** should be specified in the **terminfo** source file. In this case, initialization strings should turn on this option, if possible. If the device has a command that moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the device has no **cr** and

If it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and screen terminals. Thus, the AT&T 5320 hardcopy terminal is described as follows:

```
5320|att5320|AT&T 5320 hardcopy terminal,
    am, hc, os,
    cols#132,
    bel=^G, cr=\r, cubl=\b, cndl=\n,
    dchl=\E[P, dll=\E[M,
    ind=\n,
```

while the Lear Siegler ADM-3 is described as:

```
adm3 | lsi adm3,
    am, bel=^G, clear=^Z, cols#80, cr=^M, cubl=^H,
    cudl=^J, ind=^J, lines#24,
```

A.1.2 Parameterized Strings

Cursor addressing and other strings requiring arguments are described by a argumentized string capability with escapes in a form (%x) comparable to *printf()*. For example, to address the cursor, the **cup** capability is given, using two arguments: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory-relative cursor addressing, that can be indicated by **mrcup**.

The argument mechanism uses a stack and special % codes to manipulate the stack in the manner of Reverse Polish Notation (postfix). Typically, a sequence pushes one of the arguments onto the stack and then prints it in some format. Often more complex operations are necessary. Operations are in postfix form with the operands in the usual order. That is, to subtract 5 from the first argument, use %p1%{5}%-.

The % encodings have the following meanings:

| | |
|---------------------------------------|--|
| %% | Outputs '%'. |
| %[[:]flags][width[.precision]][doxXs] | As in <i>printf()</i> ; flags are [-+#] and space. |
| %c | Print <i>pop()</i> gives %c. |
| %p[1-9] | Push the <i>i</i> th argument. |
| %P[a-z] | Set dynamic variable [a-z] to <i>pop()</i> . |
| %g[a-z] | Get dynamic variable [a-z] and push it. |
| %P[A-Z] | Set static variable [a-z] to <i>pop()</i> . |
| %g[A-Z] | Get static variable [a-z] and push it. |
| %'c' | Push char constant <i>c</i> . |
| %{nn} | Push decimal constant <i>nn</i> . |
| %l | Push <i>strlen(pop())</i> . |
| %+ %- %* %/ %m | Arithmetic (%m is mod): push(pop <i>integer</i> ₂ op pop <i>integer</i> ₁) where <i>integer</i> ₁ represents the top of the stack. |

| | |
|---|--|
| <code>%& % %^</code> | Bit operations: <code>push(pop integer₂ op pop integer₁)</code> |
| <code>%= %> %<</code> | Logical operations: <code>push(pop integer₂ op pop integer₁)</code> |
| <code>%A %O</code> | Logical operations: and, or |
| <code>%! %~</code> | Unary operations: <code>push(op pop())</code> |
| <code>%i</code> | (For ANSI terminals) add 1 to the first argument (if one argument present), or first two arguments (if more than one argument present). |
| <code>?? expr %t thenpart %e elsepart %;</code> | If-then-else, <code>%e elsepart</code> is optional; else-if's are possible ala Algol 68: <code>?? c₁ %t b₁ %e c₂ %t b₂ %e c₃ %t b₃ %e c₄ %t b₄ %e b₅ %;</code> <code>c_i</code> are conditions, <code>b_i</code> are bodies. |

If the `-` flag is used with `%[doxXs]`, then a colon must be placed between the `%` and the `-` to differentiate the flag from the binary `%-` operator. For example: `%:-16.16s`.

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent `\E&a12c03Y` padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are zero-padded as two digits. Thus, its `cup` capability is:

```
cup=\E&a%p2%2.2dc%p1%2.2dY$<6>
```

The Micro-Term ACT-IV needs the current row and column sent preceded by a `^T`, with the row and column simply encoded in binary:

```
cup=^T%p1%c%p2%c
```

Devices that use `%c` need to be able to backspace the cursor (`cub1`), and to move the cursor up one line on the screen (`cuu1`). This is necessary because it is not always safe to transmit `\n`, `^D`, and `\r`, as the system may change or discard them. (The library functions dealing with `terminfo` set `ttty` modes so that `<tab>`s are never expanded, so `\t` is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a `<blank>` character, thus:

```
cup=\E=%p1%\s' %+%c%p2%\s' %+%c
```

After sending `\E=`, this pushes the first argument, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values), and outputs that value as a character. Then the same is done for the second argument. More complex arithmetic is possible using the stack.

A.1.3 Cursor Motions

If the terminal has a fast way to home the cursor (to very upper-left corner of screen) then this can be given as `home`; similarly, a fast way of getting to the lower left-hand corner can be given as `ll`; this may involve going up with `cuu1` from the home position, but a program should never do this itself (unless `ll` does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the `\EH` sequence on Hewlett-Packard terminals cannot be used for `home` without losing some of the other features on the terminal.)

If the device has row or column absolute-cursor addressing, these can be given as single argument capabilities `hpa` (horizontal position absolute) and `vpa` (vertical position absolute). Sometimes these are shorter than the more general two-argument sequence (as with the

Hewlett-Packard 2645) and can be used in preference to **cup**. If there are argumentized local motions (such as “move *n* spaces to the right”), these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single argument indicating how many spaces to move. These are primarily useful if the device does not have **cup**, such as the Tektronix 4025.

If the device needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals, such as the Concept, with more than one page of memory. If the device has only memory-relative cursor addressing and not screen-relative cursor addressing, a one screen-sized window must be fixed into the device for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by **terminfo**. If the **rmcup** will not restore the screen after an **smcup** sequence is output (to the state prior to outputting **smcup**) specify **nrrmc**.

A.1.4 Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the beginning of the line to the current position inclusive, leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

A.1.5 Insert/Delete Line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single argument and insert or delete that many lines can be given as **il** and **dl**.

If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two arguments: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command—the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (**ri**) followed by a delete line (**dl1**) or index (**ind**). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by **dl1** or **ind**, then the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal has non-destructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the argumentized string **wind**. The four arguments are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if

display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

A.1.6 Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations which can be described using **terminfo**. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin-Elmer Owl, make a distinction between typed and untyped <blank>s on the screen, shifting upon an insert or delete only to an untyped <blank> on the screen which is either eliminated, or expanded to two untyped <blank>s. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type **abc def** using local cursor motions (not spaces) between the **abc** and the **def**. Then position the cursor before the **abc** and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between <blank>s and untyped positions. If the **abc** shifts over to the **def** which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line *versus* multi-line insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

terminfo can describe both terminals that have an insert mode and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals that send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds padding in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an "insert mode" and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one argument, *n*, will insert *n* <blank>s.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (for example, if there is a <tab> after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one argument, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* <blank>s without moving the cursor) can be given as **ech** with one argument.

A.1.7 Highlighting, Underlining, and Visible Bells

Your device may have one or more kinds of display attributes that allow you to highlight selected characters when they appear on the screen. The following display modes (shown with the names by which they are set) may be available:

- A blinking screen (**blink**)
- Bold or extra-bright characters (**bold**)
- Dim or half-bright characters (**dim**)
- Blanking or invisible text (**invis**)
- Protected text (**prot**)
- A reverse-video screen (**rev**)
- An alternate character set (**smacs** to enter this mode and **rmacs** to exit it)

(If a command is necessary before you can enter alternate character set mode, give the sequence in **enacs** or “enable alternate-character-set” mode.) Turning on any of these modes singly may turn off other modes.

sgr0 should be used to turn off all video enhancement capabilities. It should always be specified because it represents the only way to turn off some capabilities, such as **dim** or **blink**.

Choose one display method as *standout mode* and use it to highlight error messages and other text to which you want to draw attention. Choose a form of display that provides strong contrast but that is easy on the eyes. (We recommend reverse-video plus half-bright or reverse-video alone.) The sequences to enter and exit standout mode are given as **sms0** and **rms0**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Telera 1061 do, then **xmc** should be given to tell how many spaces are left.

Sequences to begin underlining and end underlining can be specified as **smul** and **rmul**, respectively. If the device has a sequence to underline the current character and to move the cursor one space to the right (such as the Micro-Term MIME), this sequence can be specified as **uc**.

Terminals with the “magic cookie” glitch (**xmc**) deposit special “cookies” when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a newline or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), then this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. The boolean **chts** should also be given. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given, which undoes the effects of either of these modes.

If your terminal generates underlined characters by using the underline character (with no special sequences needed) even though it does not otherwise overstrike characters, then specify the capability **ul**. For devices on which a character overstriking another leaves both characters on the screen, specify the capability **os**. If overstrikes are erasable with a `<blank>`, then this

should be indicated by specifying **eo**.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine *tiparm()* arguments, called here p1 through p9. Each argument is either 0 or non-zero, as the corresponding attribute is on or off. The nine arguments are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need to be supported by **sgr**; only those for which corresponding separate attribute commands exist should be supported. For example, let's assume that the terminal in question needs the following escape sequences to turn on various modes:

| tiparm() Argument | Attribute | Escape Sequence |
|----------------------|------------|------------------|
| | none | \E[0m |
| p1 | standout | \E[0;4;7m |
| p2 | underline | \E[0;3m |
| p3 | reverse | \E[0;4m |
| p4 | blink | \E[0;5m |
| p5 | dim | \E[0;7m |
| p6 | bold | \E[0;3;4m |
| p7 | invis | \E[0;8m |
| p8 | protect | not available |
| p9 | altcharset | ^O (off) ^N (on) |

Note that each escape sequence requires a 0 to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, because this terminal has no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, such as *underline+blink*, the sequence to use would be \E[0;3;5m. The terminal doesn't have *protect* mode either, but that cannot be simulated in any way, so p8 is ignored. The *altcharset* mode is different in that it is either ^O or ^N, depending on whether it is off or on. If all modes were to be turned on, the sequence would be:

```
\E[0;3;4;5;7;8m^N
```

Now look at when different sequences are output. For example, ;3 is output when either p2 or p6 is true; that is, if either *underline* or *bol* modes are turned on. Writing out the above sequences, along with their dependencies, gives the following:

| Sequence | When to Output | terminfo Translation |
|----------|-------------------|----------------------|
| \E[0 | always | \E[0 |
| ;3 | if p2 or p6 | %%p2%p6% %;3%; |
| ;4 | if p1 or p3 or p6 | %%p1%p3% ;p6% %;4%; |
| ;5 | if p4 | %%p4%;5%; |
| ;7 | if p1 or p5 | %%p1%p5% %;7%; |
| ;8 | if p7 | %%p7%;8%; |
| m | always | m |
| ^N or ^O | if p9 ^N, else ^O | %%p9%^N%e^O%; |

Putting this all together into the **sgr** sequence gives:

```
sgr=\E[0%%p2%p6%|%;3%;%%p1%p3%|;p6%
|%;4%;%%p5%;5%;%%p1%p5%
|%;7%;%%p7%;8%;m%%p9%^N%e^O%;,
```

Remember that **sgr** and **sgr0** must always be specified.

A.1.8 Keypad

If the device has a keypad that transmits sequences when the keys are pressed, this information can also be specified. Note that it is not possible to handle devices where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, specify these sequences as **smkx** and **rmkx**. Otherwise, the keypad is assumed to always transmit.

The sequences sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1**, **kcufl1**, **kcuu1**, **kcud1**, and **khome**, respectively. If there are function keys such as **f0**, **f1**, ..., **f63**, the sequences they send can be specified as **kf0**, **kf1**, ..., **kf63**. If the first 11 keys have labels other than the default **f0** through **f10**, the labels can be given as **lf0**, **lf1**, ..., **lf10**.

The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all <tab>s), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knpp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3-by-3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3-by-3 directional pad are needed. Further keys are defined above in the capabilities list.

Strings to program function keys can be specified as **pfkey**, **pfloc**, and **pxf**. A string to program screen labels should be specified as **pln**. Each of these strings takes two arguments: a function key identifier and a string to program it with. **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local mode; and **pxf** causes the string to be transmitted to the computer. The capabilities **nlab**, **lw**, and **lh** define the number of programmable screen labels and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln**. **smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

A.1.9 Tabs and Initialization

If the device has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually <control-I>). A "backtab" command that moves leftward to the next tab stop can be given as **cbt**. By convention, if **tty** modes show that <tab>s are being expanded by the computer rather than being sent to the device, programs should not use **ht** or **cbt** (even if they are present) because the user might not have the tab stops properly set. If the device has hardware <tab>s that are initially set every *n* spaces when the device is powered up, the numeric argument **it** is given, showing the number of spaces the <tab>s are set to. This is normally used by **tput init** to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the device has tab stops that can be saved in non-volatile memory, the **terminfo** description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include: **is1**, **is2**, and **is3**, initialization strings for the device; **iprogr**, the path name of a program to be run to initialize the device; and **if**, the name of a file containing long initialization strings. These strings are expected to set the device into modes consistent with the rest of the **terminfo** description. They must be sent to the device each time the user logs in and be output in the following order: run the program **iprogr**; output **is1**; output **is2**; set the margins using **mgs**, **smgl**, and **smgr**; set the <tab>s using **tbc** and **hts**; print the file **if**; and finally output **is3**. This is usually done using the **init** option of **tput**.

Most initialization is done with **is2**. Special device modes can be set up without duplicating

strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. Sequences that do a reset from a totally unknown state can be given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is1**, **is2**, **is3**, and **if**. (The method using files, **if** and **rf**, is used for a few terminals; however, the recommended method is to use the initialization and reset strings.) These strings are output by *tput* **reset**, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs1**, **rs2**, **rs3**, and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2**, but on some terminals it causes an annoying glitch on the screen and is not normally needed because the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the <tab>s than can be described by using **tb** and **hts**, the sequence can be placed in **is2** or **if**.

Any margin can be cleared with **mgc**. (For instructions on how to specify commands to set and clear margins, see [Margins](#) (on page 371).)

A.1.10 Delays

Certain capabilities control padding in the **tty** driver. These are primarily needed by hard-copy terminals, and are used by *tput* **init** to set **tty** modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** can be used to set the appropriate delay bits to be set in the **tty** driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

A.1.11 Status Lines

If the terminal has an extra “status line” that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit H19’s 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings that go to a given column of the status line and return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The capability **tsl** takes one argument, which is the column number of the status line the cursor is to be moved to.

If escape sequences and other special commands, such as **tab**, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen (that is, **cols**). If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric argument **wsl**.

A.1.12 Line Graphics

If the device has a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the Digital VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

| Glyph Name | VT100+ Character |
|-------------------------|------------------|
| arrow pointing right | + |
| arrow pointing left | , |
| arrow pointing down | . |
| solid square block | 0 |
| lantern symbol | I |
| arrow pointing up | - |
| diamond | ` |
| checker board (stipple) | a |
| degree symbol | f |
| plus/minus | g |
| board of squares | h |
| lower right corner | j |
| upper right corner | k |
| upper left corner | l |
| lower left corner | m |
| plus | n |
| scan line 1 | o |
| horizontal line | q |
| scan line 9 | s |
| left tee (┌) | t |
| right tee (┐) | u |
| bottom tee (└) | v |
| top tee (┘) | w |
| vertical line | x |
| bullet | ~ |

The best way to describe a new device's line graphics set is to add a third column to the above table with the characters for the new device that produce the appropriate glyph when the device is in alternate-character-set mode. For example:

| Glyph Name | VT100+ Character | Character Used on New Device |
|--------------------|------------------|------------------------------|
| upper left corner | l | R |
| lower left corner | m | F |
| upper right corner | k | T |
| lower right corner | j | G |
| horizontal line | q | , |
| vertical line | x | . |

Now write down the characters left to right; for example:

```
acsc=lRmFkTjGq\,x.
```

In addition, **terminfo** lets you define multiple character sets (see [Section A.2.5](#), on page 373).

A.1.13 Color Manipulation

Most color terminals belong to one of two classes of terminal:

- **Tektronix-style**

The Tektronix method uses a set of N predefined colors (usually 8) from which an application can select “current” foreground and background colors. Thus, a terminal can support up to N colors mixed into $N*N$ color-pairs to be displayed on the screen at the same time.

- **Hewlett-Packard-style**

In the HP method, the application cannot define the foreground independently of the background, or *vice versa*. Instead, the application must define an entire color-pair at once. Up to M color-pairs, made from $2*M$ different colors, can be defined this way.

The numeric variables **colors** and **pairs** define the number of colors and color-pairs that can be displayed on the screen at the same time. If a terminal can change the definition of a color (for example, the Tektronix 4100 and 4200 series terminals), this should be specified with **ccc** (can change color). To change the definition of a color (Tektronix 4200 method), use **initc** (initialize color). It requires four arguments: color number (ranging from 0 to **colors**-1) and three RGB (red, green, and blue) values or three HLS colors (Hue, Lightness, Saturation). Ranges of RGB and HLS values are terminal-dependent.

Tektronix 4100 series terminals only use HLS color notation. For such terminals (or dual-mode terminals to be operated in HLS mode) one must define a boolean variable **hls**; that would instruct the *init_color()* functions to convert its RGB arguments to HLS before sending them to the terminal. The last three arguments to the **initc** string would then be HLS values.

If a terminal can change the definitions of colors, but uses a color notation different from RGB and HLS, a mapping to either RGB or HLS must be developed.

If the terminal supports ANSI escape sequences to set background and foreground, they should be coded as **setab** and **setaf**, respectively. If the terminal supports other escape sequences to set background and foreground, they should be coded as **setb** and **setf**, respectively. The *vidputs()* function and the refresh functions use **setab** and **setaf** if they are defined. Each of these capabilities requires one argument: the number of the color. By convention, the first eight colors (0-7) map to, in order: black, red, green, yellow, blue, magenta, cyan, white. However, color re-mapping may occur or the underlying hardware may not support these colors. Mappings for any additional colors supported by the device (that is, to numbers greater than 7) are at the discretion of the **terminfo** entry writer.

To initialize a color-pair (HP method), use **initp** (initialize pair). It requires seven arguments: the number of a color-pair (range=0 to **pairs**-1), and six RGB values: three for the foreground followed by three for the background. (Each of these groups of three should be in the order RGB.) When **initc** or **initp** are used, RGB or HLS arguments should be in the order “red, green, blue” or “hue, lightness, saturation”, respectively. To make a color-pair current, use **scp** (set color-pair). It takes one argument, the number of a color-pair.

Some terminals (for example, most color terminal emulators for PCs) erase areas of the screen with current background color. In such cases, **bce** (background color erase) should be defined. The variable **op** (original pair) contains a sequence for setting the foreground and the background colors to what they were at the terminal start-up time. Similarly, **oc** (original colors) contains a control sequence for setting all colors (for the Tektronix method) or color-pairs (for the HP method) to the values they had at the terminal start-up time.

Some color terminals substitute color for video attributes. Such video attributes should not be combined with colors. Information about these video attributes should be packed into the **ncv**

(no color video) variable. There is a one-to-one correspondence between the nine least-significant bits of that variable and the video attributes. The following table depicts this correspondence.

| Attribute | Bit Position | Decimal Value | Characteristic That Sets |
|---------------|--------------|---------------|---------------------------|
| WA_STANDOUT | 0 | 1 | sgr , parameter 1 |
| WA_UNDERLINE | 1 | 2 | sgr , parameter 2 |
| WA_REVERSE | 2 | 4 | sgr , parameter 3 |
| WA_BLINK | 3 | 8 | sgr , parameter 4 |
| WA_DIM | 4 | 16 | sgr , parameter 5 |
| WA_BOLD | 5 | 32 | sgr , parameter 6 |
| WA_INVIS | 6 | 64 | sgr , parameter 7 |
| WA_PROTECT | 7 | 128 | sgr , parameter 8 |
| WA_ALTCHARSET | 8 | 256 | sgr , parameter 9 |
| WA_HORIZONTAL | 9 | 512 | sgr1 , parameter 1 |
| WA_LEFT | 10 | 1024 | sgr1 , parameter 2 |
| WA_LOW | 11 | 2048 | sgr1 , parameter 3 |
| WA_RIGHT | 12 | 4096 | sgr1 , parameter 4 |
| WA_TOP | 13 | 8192 | sgr1 , parameter 5 |
| WA_VERTICAL | 14 | 16384 | sgr1 , parameter 6 |

When a particular video attribute should not be used with colors, set the corresponding **ncv** bit to 1; otherwise, set it to 0. To determine the information to pack into the **ncv** variable, add the decimal values corresponding to those attributes that cannot coexist with colors. For example, if the terminal uses colors to simulate reverse video (bit number 2 and decimal value 4) and bold (bit number 5 and decimal value 32), the resulting value for **ncv** will be 36 (4 + 32).

A.1.14 Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not have a pad character, specify **npc**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually <control-L>).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the argumentized string **rep**. The first argument is the character to be repeated and the second is the number of times to repeat it. Thus:

```
tiparm(repeat_char, 'x', 10)
```

is the same as xxxxxxxxxxxx.

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some systems: If the environment variable **CC** exists, all occurrences of the prototype character are replaced with the character in **CC**.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*,

dialup, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.) If the terminal is one of those supported by the virtual terminal protocol, the terminal number can be given as **vt**. A line-turn-around sequence to be transmitted before doing reads should be specified in **rft**.

If the device uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that functions can make better decisions about costs, but actual pad characters will not be transmitted. Sequences to turn on and off xon/xoff handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not ^S and ^Q , they may be specified with **xonc** and **xoffc**.

If the terminal has a “meta key” which acts as a shift key, setting the eighth bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the eighth bit is parity and it will usually be cleared. If strings exist to turn this “meta mode” on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm#0** indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as:

mc0 Print the contents of the screen.

mc4 Turn off the printer.

mc5 Turn on the printer.

When the printer is on, all text sent to the terminal will be sent to the printer. A variation, **mc5p**, takes one argument, and leaves the printer on for as many characters as the value of the argument, then turns the printer off. The argument should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

A.1.15 Special Cases

The working model used by **terminfo** fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by **terminfo**. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not have all the features of the **terminfo** model implemented.

Terminals that cannot display tilde (~) characters, such as certain Hazeltine terminals, should indicate **hz**.

Terminals that ignore a \langle linefeed \rangle immediately after an **am** wrap, such as the Concept 100, should indicate **xenl**. Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl**.

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

Those Teleray terminals whose \langle tab \rangle s turn all characters moved over to \langle blank \rangle s, should indicate **xt** (destructive \langle tab \rangle s). This capability is also taken to mean that it is not possible to position the cursor on top of a “magic cookie”. Therefore, to erase standout mode, it is necessary, instead, to use delete and insert line.

For Beehive Superbee terminals that do not transmit the <escape> or <control-C> characters, specify **xsib**, indicating that the f1 key is to be used for escape and the f2 key for <control-C>.

A.1.16 Similar Terminals

If there are two similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be canceled by placing *capability-name@* prior to the appearance of the string capability **use**. For example, the entry:

```
att4424-2|Teletype 4424 in display function group ii,
    rev@, sgr@, smul@, use=att4424,
```

defines an AT&T 04424 terminal that does not have the **rev**, **sgr**, and **smul** capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

A.2 Printer Capabilities

The **terminfo** database lets you define capabilities of printers as well as terminals. Capabilities available for printers are included in the lists in [Section 7.1.3](#) (on page 340).

A.2.1 Rounding Values

Because argumentized string capabilities work only with integer values, **terminfo** designers should create strings that expect numeric values that have been rounded. Application designers should note this and should always round values to the nearest integer before using them with a argumentized string capability.

A.2.2 Printer Resolution

A printer's resolution is defined to be the smallest spacing of characters it can achieve. In general, the horizontal and vertical resolutions are independent. Thus, the vertical resolution of a printer can be determined by measuring the smallest achievable distance between consecutive printing baselines, while the horizontal resolution can be determined by measuring the smallest achievable distance between the leftmost edges of consecutive printed, identical characters.

All printers are assumed to be capable of printing with a uniform horizontal and vertical resolution. The view of printing that **terminfo** currently presents is one of printing inside a uniform matrix: All characters are printed at fixed positions relative to each "cell" in the matrix; furthermore, each cell has the same size given by the smallest horizontal and vertical step sizes dictated by the resolution. (The cell size can be changed as will be seen later.)

Many printers are capable of "proportional printing", where the horizontal spacing depends on the size of the character last printed. **terminfo** does not make use of this capability, although it does provide enough capability definitions to allow an application to simulate proportional printing.

A printer must not only be able to print characters as close together as the horizontal and vertical resolutions suggest, but also of "moving" to a position an integral multiple of the

smallest distance away from a previous position. Thus, printed characters can be spaced apart a distance that is an integral multiple of the smallest distance, up to the length or width of a single page.

Some printers can have different resolutions depending on different “modes”. In “normal mode”, the existing **terminfo** capabilities are assumed to work on columns and lines, just like a video terminal. Thus, the old **lines** capability would give the length of a page in lines, and the **cols** capability would give the width of a page in columns. In “micro mode”, many **terminfo** capabilities work on increments of lines and columns. With some printers the micro mode may be concomitant with normal mode, so that all the capabilities work at the same time.

A.2.3 Specifying Printer Resolution

The printing resolution of a printer is given in several ways. Each specifies the resolution as the number of smallest steps per distance:

| Characteristic Number of Smallest Steps | |
|---|-----------------------------|
| orhi | Steps per inch horizontally |
| orvi | Steps per inch vertically |
| orc | Steps per column |
| orl | Steps per line |

When printing in normal mode, each character printed causes movement to the next column, except in special cases described later; the distance moved is the same as the per-column resolution. Some printers cause an automatic movement to the next line when a character is printed in the rightmost position; the distance moved vertically is the same as the per-line resolution. When printing in micro mode, these distances can be different, and may be zero for some printers.

| Automatic Motion after Printing | |
|---------------------------------|--------------------------|
| <i>Normal Mode:</i> | |
| orc | Steps moved horizontally |
| orl | Steps moved vertically |
| <i>Micro Mode:</i> | |
| mcs | Steps moved horizontally |
| mls | Steps moved vertically |

Some printers are capable of printing wide characters. The distance moved when a wide character is printed in normal mode may be different from when a regular width character is printed. The distance moved when a wide character is printed in micro mode may also be different from when a regular character is printed in micro mode, but the differences are assumed to be related: If the distance moved for a regular character is the same whether in normal mode or micro mode (**mcs=orc**), then the distance moved for a wide character is also the same whether in normal mode or micro mode. This doesn't mean the normal character distance is necessarily the same as the wide character distance, just that the distances don't change with a change in normal to micro mode. However, if the distance moved for a regular character is different in micro mode from the distance moved in normal mode (**mcs<orc**), the micro mode distance is assumed to be the same for a wide character printed in micro mode, as the table below shows.

| Automatic Motion after Printing Wide Character | |
|--|--------------------------|
| Normal Mode or Micro Mode (mcs=orc): widcs | Steps moved horizontally |
| Micro Mode (mcs<orc): mcs | Steps moved horizontally |

There may be control sequences to change the number of columns per inch (the character pitch) and to change the number of lines per inch (the line pitch). If these are used, the resolution of the printer changes, but the type of change depends on the printer:

| Changing the Character/Line Pitches | |
|-------------------------------------|---|
| cpix | Change character pitch |
| cpix | If set, cpix changes orhi ; otherwise, changes orc |
| lpix | Change line pitch |
| lpix | If set, lpix changes orvi ; otherwise, changes orl |
| chr | Change steps per column |
| cvr | Change steps per line |

The **cpix** and **lpix** string capabilities are each used with a single argument, the pitch in columns (or characters) and lines per inch, respectively. The **chr** and **cvr** string capabilities are each used with a single argument, the number of steps per column and line, respectively.

Using any of the control sequences in these strings will imply a change in some of the values of **orc**, **orhi**, **orl**, and **orvi**. Also, the distance moved when a wide character is printed, **widcs**, changes in relation to **orc**. The distance moved when a character is printed in micro mode, **mcs**, changes similarly, with one exception: if the distance is 0 or 1, then no change is assumed.

Programs that use **cpix**, **lpix**, **chr**, or **cvr** should recalculate the printer resolution (and should recalculate other values; see [Section A.2.7](#), on page 375).

| Effects of Changing the Character/Line Pitches | |
|--|--|
| Before | After |
| Using cpix with cpix clear: orhi' orc' | orhi $\text{orc} = \frac{\text{orhi}}{V_{cpix}}$ |
| Using cpix with cpix set: orhi' orc' | $\text{orhi} = \text{orc} \cdot V_{cpix}$ orc |
| Using lpix with lpix clear: orvi' orl' | orvi $\text{orl} = \frac{\text{orvi}}{V_{lpix}}$ |
| Using lpix with lpix set: orvi' orl' | $\text{orvi} = \text{orl} \cdot V_{lpix}$ orl |
| Using chr : | |

| Effects of Changing the Character/Line Pitches | |
|--|--|
| Before | After |
| orhi' orc' | orhi V_{chr} |
| Using cvr: orvi' orl' | orvi V_{cvr} |
| Using cpi or chr: widcs' mcs' | $\text{widcs} = \text{widcs}' \frac{\text{orc}}{\text{orc}'}$ $\text{mcs} = \text{mcs}' \frac{\text{orc}}{\text{orc}'}$ |

V_{cpi} , V_{lpi} , V_{chr} , and V_{cvr} are the arguments used with **cpi**, **lpi**, **chr**, and **cvr**, respectively. The prime marks (' ') indicate the old values.

A.2.4 Capabilities that Cause Movement

In the following descriptions, “movement” refers to the motion of the “current position”. With video terminals this would be the cursor; with some printers, this is the carriage position. Other printers have different equivalents. In general, the current position is where a character would be displayed if printed.

terminfo has string capabilities for control sequences that cause movement a number of full columns or lines. It also has equivalent string capabilities for control sequences that cause movement a number of smaller steps.

| String Capabilities for Motion | |
|--------------------------------|------------------------------|
| mcub1 | Move 1 step left |
| mcuf1 | Move 1 step right |
| mcuu1 | Move 1 step up |
| mcud1 | Move 1 step down |
| mcub | Move N steps left |
| mcuf | Move N steps right |
| mcuu | Move N steps up |
| mcud | Move N steps down |
| mhpa | Move N steps from the left |
| mvpa | Move N steps from the top |

The latter six strings are each used with a single argument, N .

Sometimes the motion is limited to less than the width or length of a page. Also, some printers don't accept absolute motion to the left of the current position. **terminfo** has capabilities for specifying these limits.

| Limits to Motion | |
|------------------|---|
| mjump | Limit on use of mcub1 , mcuf1 , mcuu1 , mcud1 |
| maddr | Limit on use of mhpa , mvpa |
| xhpa | If set, hpa and mhpa can't move left |
| xvpa | If set, vpa and mvpa can't move up |

If a printer needs to be in a “micro mode” for the motion capabilities described above to work, there are string capabilities defined to contain the control sequence to enter and exit this mode.

A boolean is available for those printers where using a <carriage-return> causes an automatic return to normal mode.

| Entering/Exiting Micro Mode | |
|-----------------------------|----------------------------------|
| smicm | Enter micro mode |
| rmicm | Exit micro mode |
| crxm | Using cr exits micro mode |

The movement made when a character is printed in the rightmost position varies among printers. Some make no movement, some move to the beginning of the next line, others move to the beginning of the same line. **terminfo** has boolean capabilities for describing all three cases.

| What Happens After Character Printed in Rightmost Position | |
|--|--|
| sam | Automatic move to beginning of same line |

Some printers can be put in a mode where the normal direction of motion is reversed. This mode can be especially useful when there are no capabilities for leftward or upward motion, because those capabilities can be built from the motion reversal capability and the rightward or downward motion capabilities. It is best to leave it up to an application to build the leftward or upward capabilities, though, and not enter them in the **terminfo** database. This allows several reverse motions to be strung together without intervening wasted steps that leave and reenter reverse mode.

| Entering/Exiting Reverse Modes | |
|--|-------------------------------------|
| slm | Reverse sense of horizontal motions |
| rlm | Restore sense of horizontal motions |
| sum | Reverse sense of vertical motions |
| rum | Restore sense of vertical motions |
| <i>While sense of horizontal motions reversed:</i> | |
| mcub1 | Move 1 step right |
| mcuf1 | Move 1 step left |
| mcub | Move <i>N</i> steps right |
| mcuf | Move <i>N</i> steps left |
| cub1 | Move 1 column right |
| cuf1 | Move 1 column left |
| cub | Move <i>N</i> columns right |
| cuf | Move <i>N</i> columns left |
| <i>While sense of vertical motions reversed:</i> | |
| mcuu1 | Move 1 step down |
| mcud1 | Move 1 step up |
| mcuu | Move <i>N</i> steps down |
| mcud | Move <i>N</i> steps up |
| cuu1 | Move 1 line down |
| cud1 | Move 1 line up |
| cuu | Move <i>N</i> lines down |
| cud | Move <i>N</i> lines up |

The reverse motion modes should not affect the **mvpa** and **mhpa** absolute motion capabilities. The reverse vertical motion mode should, however, also reverse the action of the line “wrapping” that occurs when a character is printed in the right-most position. Thus, printers that have the standard **terminfo** capability **am** defined should experience motion to the beginning of the previous line when a character is printed in the rightmost position in reverse vertical motion mode.

The action when any other motion capabilities are used in reverse motion modes is not defined; thus, programs must exit reverse motion modes before using other motion capabilities.

Two miscellaneous capabilities complete the list of motion capabilities. One of these is needed for printers that move the current position to the beginning of a line when certain control characters, such as <line-feed> or <form-feed>, are used. The other is used for the capability of suspending the motion that normally occurs after printing a character.

| Miscellaneous Motion Strings | |
|------------------------------|--|
| docr | List of control characters causing cr |
| zerom | Prevent auto motion after printing next single character |

Margins

terminfo provides two strings for setting margins on terminals: one for the left and one for the right margin. Printers, however, have two additional margins, for the top and bottom margins of each page. Furthermore, some printers require not using motion strings to move the current position to a margin and then fixing the margin there, but require the specification of where a margin should be regardless of the current position. Therefore, **terminfo** offers six additional strings for defining margins with printers.

| Setting Margins | |
|-----------------|-------------------------------------|
| smgl | Set left margin at current column |
| smgr | Set right margin at current column |
| smgb | Set bottom margin at current line |
| smt | Set top margin at current line |
| smgbp | Set bottom margin at line <i>N</i> |
| smglp | Set left margin at column <i>N</i> |
| smgrp | Set right margin at column <i>N</i> |
| smgtp | Set top margin at line <i>N</i> |

The last four strings are used with one or more arguments that give the position of the margin or margins to set. If both of **smglp** and **smgrp** are set, each is used with a single argument, *N*, that gives the column number of the left and right margin, respectively. If both of **smgtp** and **smgbp** are set, each is used to set the top and bottom margin, respectively: **smgtp** is used with a single argument, *N*, the line number of the top margin; however, **smgbp** is used with two arguments, *N* and *MM*, that give the line number of the bottom margin, the first counting from the top of the page and the second counting from the bottom. This accommodates the two styles of specifying the bottom margin in different manufacturers' printers. When coding a **terminfo** entry for a printer that has a settable bottom margin, only the first or second argument should be used, depending on the printer. When writing an application that uses **smgbp** to set the bottom margin, both arguments must be given.

If only one of **smglp** and **smgrp** is set, then it is used with two arguments, the column number of the left and right margins, in that order. Likewise, if only one of **smgtp** and **smgbp** is set, then it is used with two arguments that give the top and bottom margins, in that order, counting from the top of the page. Thus, when coding a **terminfo** entry for a printer that requires setting both left and right or top and bottom margins simultaneously, only one of **smgl** and **smgrp** or **smgtp** and **smgbp** should be defined; the other should be left blank. When writing an application that uses these string capabilities, the pairs should be first checked to see if each in the pair is set or only one is set, and should then be used accordingly.

In counting lines or columns, line zero is the top line and column zero is the left-most column. A zero value for the second argument with **smgbp** means the bottom line of the page.

All margins can be cleared with **mgc**.

Shadows, Italics, Wide Characters, Superscripts, Subscripts

Five sets of strings describe the capabilities printers have of enhancing printed text.

| Enhanced Printing | |
|-------------------|--|
| sshm | Enter shadow-printing mode |
| rsh | Exit shadow-printing mode |
| sitm | Enter italicizing mode |
| ritm | Exit italicizing mode |
| swidm | Enter wide character mode |
| rwidm | Exit wide character mode |
| ssupm | Enter superscript mode |
| rsupm | Exit superscript mode |
| supcs | List of characters available as superscripts |
| ssubm | Enter subscript mode |
| rsubm | Exit subscript mode |
| subcs | List of characters available as subscripts |

If a printer requires the **sshm** control sequence before every character to be shadow-printed, the **rsh** string is left blank. Thus, programs that find a control sequence in **sshm** but none in **rsh** should use the **sshm** control sequence before every character to be shadow-printed; otherwise, the **sshm** control sequence should be used once before the set of characters to be shadow-printed, followed by **rsh**. The same is also true of each of the **sitm/ritm**, **swidm/rwidm**, **ssupm/rsupm**, and **ssubm/rsubm** pairs.

terminfo also has a capability for printing emboldened text (**bold**). While shadow printing and emboldened printing are similar in that they “darken” the text, many printers produce these two types of print in slightly different ways. Generally, emboldened printing is done by overstriking the same character one or more times. Shadow printing likewise usually involves overstriking, but with a slight movement up and/or to the side so that the character is “fatter”.

It is assumed that enhanced printing modes are independent modes, so that it would be possible, for instance, to shadow print italicized subscripts.

As mentioned earlier, the amount of motion automatically made after printing a wide character should be given in **widcs**.

If only a subset of the printable ASCII characters can be printed as superscripts or subscripts, they should be listed in **supcs** or **subcs** strings, respectively. If the **ssupm** or **ssubm** strings contain control sequences, but the corresponding **supcs** or **subcs** strings are empty, it is assumed that all printable ASCII characters are available as superscripts or subscripts.

Automatic motion made after printing a superscript or subscript is assumed to be the same as for regular characters. Thus, for example, printing any of the following three examples results in equivalent motion:

B_i B_i Bⁱ

Note that the existing **msgr** boolean capability describes whether motion control sequences can be used while in “standout mode”. This capability is extended to cover the enhanced printing modes added here. **msgr** should be set for those printers that accept any motion control sequences without affecting shadow, italicized, widened, superscript, or subscript printing. Conversely, if **msgr** is not set, a program should end these modes before attempting any motion.

A.2.5 Alternate Character Sets

In addition to allowing you to define line graphics (described in [Section A.1.12](#), on page 361), **terminfo** lets you define alternate character sets. The following capabilities cover printers and terminals with multiple selectable or definable character sets:

| Alternate Character Sets | |
|--------------------------|--|
| scs | Select character set <i>N</i> |
| sbsd | Start definition of character set <i>N</i> , <i>M</i> characters |
| defc | Define character <i>A</i> , <i>B</i> dots wide, descender <i>D</i> |
| rcsd | End definition of character set <i>N</i> |
| csnm | List of character set names |
| daisy | Printer has manually changed print-wheels |

The **scs**, **rcsd**, and **csnm** strings are used with a single argument, *N*, a number from 0 to 63 that identifies the character set. The **sbsd** string is also used with the argument *N* and another, *M*, that gives the number of characters in the set. The **defc** string is used with three arguments: *A* gives the ASCII code representation for the character, *B* gives the width of the character in dots, and *D* is zero or one depending on whether the character is a “descender” or not. The **defc** string is also followed by a string of “image-data” bytes that describe how the character looks (see below).

Character set 0 is the default character set present after the printer has been initialized. Not every printer has 64 character sets, of course; using **scs** with an argument that doesn’t select an available character set should cause a null pointer to be returned by *tiparm()*.

If a character set has to be defined before it can be used, the **sbsd** control sequence is to be used before defining the character set, and the **rcsd** is to be used after. They should also cause a NULL pointer to be returned by *tiparm()* when used with an argument *N* that doesn’t apply. If a character set still has to be selected after being defined, the **scs** control sequence should follow the **rcsd** control sequence. By examining the results of using each of the **scs**, **sbsd**, and **rcsd** strings with a character set number in a call to *tiparm()*, a program can determine which of the three are needed.

Between use of the **sbsd** and **rcsd** strings, the **defc** string should be used to define each character. To print any character on printers covered by **terminfo**, the ASCII code is sent to the printer. This is true for characters in an alternate set as well as “normal” characters. Thus, the definition of a character includes the ASCII code that represents it. In addition, the width of the character in dots is given, along with an indication of whether the character should descend below the print line (such as the lowercase letter “g” in most character sets). The width of the character in dots also indicates the number of image-data bytes that will follow the **defc** string. These image-data bytes indicate where in a dot-matrix pattern ink should be applied to “draw” the character; the number of these bytes and their form are defined in [Section A.2.6](#) (on page 374).

It is easiest for the creator of **terminfo** entries to refer to each character set by number; however, these numbers will be meaningless to the application developer. The **csnm** string alleviates this problem by providing names for each number.

When used with a character set number in a call to *tiparm()*, the **csnm** string will produce the equivalent name. These names should be used as a reference only. No naming convention is implied, although anyone who creates a **terminfo** entry for a printer should use names consistent with the names found in user documents for the printer. Application developers should allow a user to specify a character set by number (leaving it up to the user to examine the **csnm** string to determine the correct number), or by name, where the application examines the **csnm** string to determine the corresponding character set number.

These capabilities are likely to be used only with dot-matrix printers. If they are not available, the strings should not be defined. For printers that have manually changed print-wheels or font cartridges, the boolean **daisy** is set.

A.2.6 Dot-Matrix Graphics

Dot-matrix printers typically have the capability of reproducing raster graphics images. Three numeric capabilities and three string capabilities help a program draw raster-graphics images independent of the type of dot-matrix printer or the number of pins or dots the printer can handle at one time.

| Dot-Matrix Graphics | |
|---------------------|--|
| npins | Number of pins, N , in print-head |
| spinv | Spacing of pins vertically in pins per inch |
| spinh | Spacing of dots horizontally in dots per inch |
| porder | Matches software bits to print-head pins |
| sbim | Start printing bit image graphics, B bits wide |
| rbim | End printing bit image graphics |

The **sbim** string is used with a single argument, B , the width of the image in dots.

The model of dot-matrix or raster-graphics that **terminfo** presents is similar to the technique used for most dot-matrix printers: each pass of the printer's print-head is assumed to produce a dot-matrix that is N dots high and B dots wide. This is typically a wide, squat, rectangle of dots. The height of this rectangle in dots will vary from one printer to the next; this is given in the **npins** numeric capability. The size of the rectangle in fractions of an inch will also vary; it can be deduced from the **spinv** and **spinh** numeric capabilities. With these three values an application can divide a complete raster-graphics image into several horizontal strips, perhaps interpolating to account for different dot spacing vertically and horizontally.

The **sbim** and **rbim** strings start and end a dot-matrix image, respectively. The **sbim** string is used with a single argument that gives the width of the dot-matrix in dots. A sequence of "image-data bytes" are sent to the printer after the **sbim** string and before the **rbim** string. The number of bytes is an integral multiple of the width of the dot-matrix; the multiple and the form of each byte is determined by the **porder** string as described below.

The **porder** string is a comma-separated list of pin numbers optionally followed by a numerical offset. The offset, if given, is separated from the list with a semicolon. The position of each pin number in the list corresponds to a bit in an 8-bit data byte. The pins are numbered consecutively from 1 to **npins**, with 1 being the top pin. Note that the term "pin" is used loosely here; "ink-jet" dot-matrix printers don't have pins, but can be considered to have an equivalent method of applying a single dot of ink to paper. The bit positions in **porder** are in groups of 8, with the first position in each group the most significant bit and the last position the least significant bit. An application produces 8-bit bytes in the order of the groups in **porder**.

An application computes the "image-data bytes" from the internal image, mapping vertical dot positions in each print-head pass into 8-bit bytes, using a 1 bit where ink should be applied and 0 where no ink should be applied. This can be reversed (0 bit for ink, 1 bit for no ink) by giving a negative pin number. If a position is skipped in **porder**, a 0 bit is used. If a position has a lowercase 'x' instead of a pin number, a 1 bit is used in the skipped position. For consistency, a lowercase 'o' can be used to represent a 0 filled, skipped bit. There must be a multiple of 8-bit positions used or skipped in **porder**; if not, low-order bits of the last byte are set to 0. The offset, if given, is added to each data byte; the offset can be negative.

Some examples may help clarify the use of the **porder** string. The AT&T 470, AT&T 475, and

C.Itoh 8510 printers provide eight pins for graphics. The pins are identified top to bottom by the 8 bits in a byte, from least significant to most. The **porder** strings for these printers would be **8,7,6,5,4,3,2,1**. The AT&T 478 and AT&T 479 printers also provide eight pins for graphics. However, the pins are identified in the reverse order. The **porder** strings for these printers would be **1,2,3,4,5,6,7,8**. The AT&T 5310, AT&T 5320, Digital LA100, and Digital LN03 printers provide six pins for graphics. The pins are identified top to bottom by the decimal values 1, 2, 4, 8, 16, and 32. These correspond to the low six bits in an 8-bit byte, although the decimal values are further offset by the value 63. The **porder** string for these printers would be **,,6,5,4,3,2,1;63**, or alternately **0,0,6,5,4,3,2,1;63**.

A.2.7 Effect of Changing Printing Resolution

If the control sequences to change the character pitch or the line pitch are used, the pin or dot spacing may change:

| Changing the Character/Line Pitches | |
|-------------------------------------|---|
| cpi cpix | Change character pitch If set, cpi changes spinh |
| lpi lpix | Change line pitch If set, lpi changes spinv |

Programs that use **cpi** or **lpi** should recalculate the dot spacing:

| Effects of Changing the Character/Line Pitches | |
|---|---|
| Before | After |
| Using cpi with cpix clear: spinh' | spinh |
| Using cpi with cpix set: spinh' | $\text{spinh} = \text{spinh}' \cdot \frac{\text{orhi}}{\text{orhi}'}$ |
| Using lpi with lpix clear: spinv' | spinv |
| Using lpi with lpix set: spinv' | $\text{spinv} = \text{spinv}' \cdot \frac{\text{orhi}}{\text{orhi}'}$ |
| Using chr : spinh' | spinh |
| Using cvr : spinv' | spinv |

orhi' and **orhi** are the values of the horizontal resolution in steps per inch, before using **cpi** and after using **cpi**, respectively. Likewise, **orvi'** and **orv** are the values of the vertical resolution in steps per inch, before using **lpi** and after using **lpi**, respectively. Thus, the changes in the dots per inch for dot-matrix graphics follow the changes in steps per inch for printer resolution.

A.2.8 Print Quality

Many dot-matrix printers can alter the dot spacing of printed text to produce *near-letter-quality* printing or *draft-quality* printing. It is important to be able to choose one or the other because the rate of printing generally decreases as the quality improves. Three strings describe these capabilities:

| Print Quality | |
|---------------|-------------------------------|
| snlq | Set near-letter quality print |
| snrmq | Set normal quality print |
| sdrfq | Set draft quality print |

The capabilities are listed in decreasing levels of quality. If a printer doesn't have all three levels, the respective strings should be left blank.

A.2.9 Printing Rate and Buffer Size

Because there is no standard protocol that can be used to keep a program synchronized with a printer, and because modern printers can buffer data before printing it, a program generally cannot determine at any time what has been printed. Two numeric capabilities can help a program estimate what has been printed.

| Print Rate/Buffer Size | |
|------------------------|---|
| cps | Nominal print rate in characters per second |
| bufsz | Buffer capacity in characters |

cps is the nominal or average rate at which the printer prints characters; if this value is not given, the rate should be estimated at one-tenth the prevailing baud rate. **bufsz** is the maximum number of subsequent characters buffered before the guaranteed printing of an earlier character, assuming proper flow control has been used. If this value is not given it is assumed that the printer does not buffer characters, but prints them as they are received.

As an example, if a printer has a 1000-character buffer, then sending the letter 'a' followed by 1000 additional characters is guaranteed to cause the letter 'a' to print. If the same printer prints at the rate of 100 characters per second, then it should take 10 seconds to print all the characters in the buffer, less if the buffer is not full. By keeping track of the characters sent to a printer, and knowing the print rate and buffer size, a program can synchronize itself with the printer.

Note that most printer manufacturers advertise the maximum print rate, not the nominal print rate. A good way to get a value to put in for **cps** is to generate a few pages of text, count the number of printable characters, and then see how long it takes to print the text.

Applications that use these values should recognize the variability in the print rate. Straight text, in short lines, with no embedded control sequences will probably print at close to the advertised print rate and probably faster than the rate in **cps**. Graphics data with a lot of control sequences, or very long lines of text, will print at well below the advertised rate and below the rate in **cps**. If the application is using **cps** to decide how long it should take a printer to print a block of text, the application should pad the estimate. If the application is using **cps** to decide how much text has already been printed, it should shrink the estimate. The application will thus err in favor of the user, who wants, above all, to see all the output in its correct place.

A.3 Selecting a Terminal

If the environment variable *TERMINFO* is defined, any program using Curses checks for a local terminal definition before checking in the standard place. For example, on implementations which use the traditional directory layout for the **terminfo** data, if *TERM* is set to **att4424**, then the compiled terminal definition is found in by default the path:

a/att4424

within an implementation-specific directory.

(The **a** is copied from the first letter of **att4424** to avoid creation of huge directories.) However, if *TERMINFO* is set to **\$HOME/myterms**, Curses first checks:

\$HOME/myterms/a/att4424

If that fails, it then checks the default pathname.

This is useful for developing experimental definitions or when write permission in the implementation-defined default database is not available.

If the *LINES* and *COLUMNS* environment variables are set, or if the program is executing in a window environment, line and column information in the environment will override information read by **terminfo**. The *use_env()* function can be used to override this default behavior.

A.4 Application Usage

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in **terminfo** and to build up a description gradually, using partial descriptions with a screen-oriented editor, to check that they are correct. To easily test a new terminal description the environment variable *TERMINFO* can be set to the pathname of a directory containing the compiled description, and programs will look there rather than in the **terminfo** database.

A.4.1 Conventions for Device Aliases

Every device must be assigned a name, such as **vt100**. Device names (except the long name) should be chosen using the following conventions. The name should not contain hyphens because hyphens are reserved for use when adding suffixes that indicate special modes.

These special modes may be modes that the hardware can be in, or user preferences. To assign a special mode to a particular device, append a suffix consisting of a hyphen and an indicator of the mode to the device name. For example, the **-w** suffix means *wide mode*; when specified, it allows for a width of 132 columns instead of the standard 80 columns. Therefore, if you want to use a vt100 device set to wide mode, name the device **vt100-w**. Use the following suffixes where possible:

| Suffix | Meaning | Example |
|--------|--|-----------|
| -w | Wide mode (more than 80 columns) | 5410-w |
| -am | With automatic margins (usually default) | vt100-am |
| -nam | Without automatic margins | vt100-nam |
| -n | Number of lines on the screen | 2300-40 |
| -na | No arrow keys (leave them in local) | c100-na |
| -np | Number of pages of memory | c100-4p |
| -rv | Reverse video | 4415-rv |

A.4.2 Variations of Terminal Definitions

It is implementation-defined how the entries in **terminfo** may be created.

There is more than one way to write a **terminfo** entry. A minimal entry may permit applications to use Curses to operate the terminal. If the entry is enhanced to describe more of the terminal's capabilities, applications can use Curses to invoke those features, and can take advantages of optimizations within Curses and thus operate more efficiently. For most terminals, an optimal **terminfo** entry has already been written.

Glossary

background

A property of a window that specifies a character (the background character) and a rendition to be used in a variety of situations. See [Section 3.3.6](#) (on page 17).

Curses window

Data structures, which can be thought of as two-dimensional arrays of characters that represent screen displays. These data structures are manipulated with Curses functions.

cursor position

The line and column position on the screen denoted by the terminal's cursor.

empty wide-character string

A wide-character string whose first element is a null wide-character code.

erase character

A special input character that deletes the last character in the current line, if there is one.

kill character

A special input character that deletes all data in the current line, if there are any.

null chtype

A **chtype** with all bits set to zero.

null wide-character code

A wide-character code with all bits set to zero.

pad

A window that is not necessarily associated with a viewable part of a screen.

parent window

A window that has subwindows or derived windows associated with it.

rendition

The rendition of a character displayed on the screen is its attributes and a color pair.

SCREEN

An opaque Curses data type that is associated with the display screen.

subwindow

A window, created within another window, but positioned relative to that other window. Changes made to a subwindow do not affect its parent window. A derived window differs from a subwindow only in that it is positioned relative to the origin of its parent window. Changes to a parent window will affect both subwindows and derived windows.

touch

To set a flag in a window that indicates that the information in the window could differ from that displayed on the terminal device.

wide-character code (C language)

An integer value corresponding to a single graphic symbol or control code.

wide-character string

A contiguous sequence of wide-character codes terminated by and including the first null wide-character code.

window

A two-dimensional array of characters representing all or part of the terminal screen. The term *window* in this document means one of the data structures maintained by the Curses implementation, unless specified otherwise. (This document does not define the interaction between the Curses implementation and other windowing system paradigms.)

window hierarchy

The aggregate of a parent window and all of its subwindows and derived windows.

Index

| | |
|--------------------------------------|----------|
| -w suffix | 377 |
| <curses.h> | 306 |
| <term.h> | 320 |
| <unctrl.h> | 321 |
| @..... | 366 |
| XBD specification | |
| relationship to | 13 |
| XSH specification | |
| relationship to | 13 |
| _w infix | 25 |
| _XOPEN_SOURCE | 10 |
| acsc | 361 |
| add | |
| effect on straddling character | 20 |
| resulting rendition..... | 21 |
| add function..... | 18 |
| addch() | 36 |
| addchnstr..... | 37 |
| addchstr()..... | 37 |
| addnstr() | 38 |
| addnwstr() | 40 |
| addstr | 38 |
| addwstr..... | 40 |
| add_wch()..... | 33 |
| add_wchnstr()..... | 34 |
| add_wchstr..... | 34 |
| adjustment of cursor position..... | 19 |
| advertised print rate | 376 |
| advisory delay | 350 |
| alias | |
| in terminfo..... | 337 |
| alternate character set..... | 358, 373 |
| line drawing..... | 361 |
| alternate keypad..... | 360 |
| am | 353 |
| ignoring linefeed after | 365 |
| ancestor..... | 15 |
| Ann Arbor 4080 (example) | 355 |
| ANSI foreground/background | 363 |
| application consideration..... | 27 |
| area clear..... | 356 |
| arrow keys..... | 360 |
| asterisk | |
| in terminfo..... | 350 |
| AT&T 4410v1 | |
| line drawing..... | 361 |
| AT&T 470/475 | 374 |

| | |
|--|-----------------|
| AT&T 5320 (example) | 354 |
| AT&T 610 (example) | 349 |
| attribute | 16 |
| attroff() | 43 |
| attron | 43 |
| attrset | 43 |
| attr_get() | 41 |
| attr_off | 41 |
| attr_on | 41 |
| attr_set | 41 |
| audible signal | 353 |
| automatic margin | 353 |
| automatic motion | 372 |
| auxiliary printer control | 365 |
| background | 18 , 379 |
| background character | 18 |
| implicit use | 21 |
| background color | 363 |
| backslash | |
| use in terminfo | 337 |
| backslash in terminfo | 350 |
| backspace | |
| special processing | 21 |
| basic capability | 353 |
| baud rate, versus printer throughput | 376 |
| baudrate() | 44 |
| bce | 363 |
| Beehive Superbee | 366 |
| beep() | 45 |
| bel | 353 |
| delays | 350 |
| bell | 353 |
| visible | 358 |
| bidirectional writing | 3 |
| bkgd() | 46 |
| bkgdset | 46 |
| bkgrnd() | 48 |
| bkgrndset | 48 |
| blanking text | 358 |
| blink | 358 |
| blinking screen | 358 |
| block cursor | 358 |
| block mode | 27 |
| bold | 358 |
| printing | 372 |
| boolean capability | 338 |
| border() | 50 |
| eliminates straddling characters | 20 |
| border_set() | 52 |
| box drawing | 353 |
| box() | 54 |
| box_set() | 55 |

| | |
|-----------------------------------|-----------|
| brightness of character | 358 |
| buffer size | 376 |
| bufsz | 376 |
| bw | 353 |
| C.Itoh 8510..... | 374 |
| calculating print rate..... | 376 |
| can | 4 |
| can_change_color()..... | 56 |
| capability of device | 337 |
| capability, device | 338 |
| carriage-return | |
| special processing..... | 21 |
| cbreak()..... | 59 |
| cbt | 360 |
| CC environment variable..... | 364 |
| ccc | 363 |
| change | |
| affecting subwindow | 14 |
| change resolution | 375 |
| character | |
| replacement..... | 18 |
| resulting rendition..... | 21 |
| straddling | 20 |
| character insert/delete | 19, 357 |
| character set | |
| alternate | 358, 373 |
| as sub/superscript | 372 |
| line drawing..... | 361 |
| name..... | 373 |
| character spacing..... | 366 |
| chgat()..... | 60 |
| chr | 368 |
| recalculate resolution after..... | 368 |
| chts | 358 |
| civis..... | 358 |
| clear | 353 |
| clear screen..... | 353 |
| clear to end-of-line | 356 |
| clear()..... | 61 |
| clearok() | 62 |
| clipping of window | 14 |
| clrtoebot()..... | 64 |
| clrtoeol() | 65 |
| cmdch..... | 364 |
| cnorm | 358 |
| codeset | 1 |
| color..... | 16 |
| color manipulation..... | 363 |
| COLORS | 30 |
| colors | 363 |
| color_content..... | 56 |
| color_content() | 66 |

| | |
|-------------------------------------|----------|
| COLOR_PAIRS..... | 30 |
| color_set..... | 41 |
| color_set() | 67 |
| COLS..... | 31 |
| cols..... | 353 |
| status line..... | 361 |
| column | |
| orphaned | 18 |
| COLUMNS..... | 377 |
| comma | |
| after last entry in terminfo | 349 |
| use in terminfo..... | 337 |
| command character..... | 364 |
| comment in terminfo | 351 |
| compilation environment..... | 10 |
| complex character | 17 |
| function naming | 25 |
| Concept (example) | 356 |
| Concept 100 | |
| ignoring linefeed after wrap..... | 365 |
| Concept 100 (example) | 357 |
| conformance..... | 3 |
| conventions, lexical..... | 339 |
| cookie | 358 |
| coordinate pair..... | 18 |
| copywin()..... | 68 |
| cpi | 375 |
| recalculate resolution after..... | 368 |
| cpix | 375 |
| cpi[x]..... | 368 |
| cps..... | 376 |
| cr..... | 353 |
| delays | 361 |
| crxm..... | 370 |
| csnm | 373 |
| csr..... | 356 |
| cub | 355 |
| cub1 | 353, 355 |
| delays | 361 |
| cud | 355 |
| cuf..... | 355 |
| cuf1 | 353 |
| cup | 354 |
| current or specified position..... | 25 |
| current or specified window | 24 |
| current position | 369 |
| curscr..... | 71 |
| Curses | 1 |
| Curses window..... | 379 |
| cursor | |
| actual position | 18 |
| analogue in printing terminal | 369 |

| | |
|-------------------------------------|-----------|
| appearance of..... | 358 |
| cursor addressing..... | 354 |
| cursor movement..... | 353 |
| relocation..... | 19 |
| within row or column..... | 355 |
| cursor position..... | 18, 379 |
| at insert/delete..... | 19 |
| curs_set()..... | 70 |
| cur_term..... | 69 |
| cuu..... | 355 |
| cuu1..... | 355 |
| cu[b/d/f/u][1]..... | 370 |
| cvr..... | 368 |
| recalculate resolution after..... | 368 |
| cvvis..... | 358 |
| da..... | 356 |
| daisy..... | 373 |
| darkened printing..... | 372 |
| data types..... | 12 |
| database, terminfo..... | 337 |
| Datamedia (example)..... | 357 |
| db..... | 356 |
| dch..... | 357 |
| dch1..... | 357 |
| defc..... | 373 |
| definition, sharing..... | 366 |
| def_prog_mode()..... | 72 |
| def_shell_mode..... | 72 |
| delay..... | 350, 361 |
| delay mode..... | 24 |
| delay_output()..... | 75 |
| delch()..... | 76 |
| delete | |
| effect on straddling character..... | 20 |
| delete/insert character..... | 357 |
| delete/insert line..... | 356 |
| deleteln()..... | 77 |
| deletion..... | 20 |
| delscreen()..... | 78 |
| delwin()..... | 79 |
| del_curterm()..... | 73 |
| derived window..... | 15 |
| derwin()..... | 80 |
| description of device..... | 337 |
| destructive scrolling..... | 356 |
| destructive tab..... | 365 |
| device capability..... | 337 |
| device name..... | 377 |
| dialup terminal..... | 364 |
| Digital LA100, LN03..... | 374 |
| dim..... | 358 |
| direct cursor addressing..... | 354 |

| | |
|------------------------------------|-----------|
| dl..... | 356 |
| dl1..... | 356 |
| docr..... | 371 |
| dot-matrix graphics | 374 |
| doupdate()..... | 82 |
| draft-quality..... | 376 |
| drawing a box..... | 353 |
| dsl..... | 361 |
| dupwin()..... | 83 |
| EC..... | 5 |
| ech..... | 357 |
| echo processing | 24 |
| echo()..... | 84 |
| echochar()..... | 86 |
| echo_wchar()..... | 85 |
| ed..... | 356 |
| eighth bit..... | 365 |
| el..... | 356 |
| el1..... | 356 |
| empty wide-character string | 379 |
| emulator, terminal..... | 363 |
| en..... | 355 |
| enacs..... | 358 |
| end-of-line | |
| truncation/wrapping | 19 |
| endwin()..... | 87 |
| enhanced character set | 2 |
| enhancement, turn off..... | 358 |
| eo..... | 358 |
| erase..... | 61 |
| erase character..... | 379 |
| erase to end-of-line..... | 356 |
| erase()..... | 88 |
| erasechar()..... | 89 |
| erasewchar..... | 89 |
| error numbers..... | 13 |
| escape in terminfo..... | 350 |
| escape sequence..... | 14 |
| eslok..... | 361 |
| estimating printer throughput..... | 376 |
| et..... | 355 |
| extension | |
| EC..... | 5 |
| OB..... | 5 |
| extra line of screen..... | 361 |
| extra-bright character..... | 358 |
| ff..... | 364 |
| delays..... | 361 |
| filter()..... | 91 |
| first line in terminfo..... | 337 |
| flag, touched..... | 14 |
| flash..... | 358 |

| | |
|------------------------------------|------------|
| delays | 350 |
| flash() | 92 |
| flashing screen | 358 |
| flow control | 365 |
| flushinp() | 93 |
| foreground color | 363 |
| form feed | 371 |
| format of entries | 6 |
| format of terminfo | 337 |
| fsl | 361 |
| function naming | 24 |
| functions | |
| implementation | 9 |
| use | 9 |
| generic terminal description | 364 |
| getbegyx() | 97 |
| getbkgd | 46 |
| getbkgd() | 99 |
| getbkgrnd | 48 |
| getbkgrnd() | 100 |
| getcchar() | 101 |
| getch() | 102 |
| getmaxyx | 97 |
| getmaxyx() | 104 |
| getnstr() | 107 |
| getn_wstr() | 105 |
| getparyx | 97 |
| getparyx() | 109 |
| getstr | 107 |
| getstr() | 110 |
| getwin() | 111 |
| getyx | 97 |
| getyx() | 112 |
| get_wch() | 94 |
| get_wstr | 105 |
| get_wstr() | 96 |
| glitch, magic cookie | 358 |
| glyph | 20 |
| gn | 364 |
| grammar | 338 |
| graphic rendition, setting | 359 |
| graphics, dot-matrix | 374 |
| graphics, line-drawing | 361 |
| half line cursor movement | 364 |
| half-bright character | 358 |
| halfdelay() | 113 |
| has_colors | 56 |
| has_colors() | 114 |
| has_ic() | 115 |
| has_il | 115 |
| Hazeltine | 365 |
| hc | 353 |

| | |
|--|------------|
| hd..... | 364 |
| header line in terminfo..... | 337 |
| headers..... | 305 |
| Heathkit H19 (example)..... | 361 |
| Hewlett-Packard | |
| model of color specification..... | 363 |
| Hewlett-Packard 2621 | |
| keypad..... | 360 |
| magic cookie glitch..... | 358 |
| Hewlett-Packard 2645 (example)..... | 355 |
| high-order bit, setting..... | 365 |
| highlighting..... | 358 |
| hline()..... | 116 |
| hline_set()..... | 117 |
| hls..... | 363 |
| home..... | 355 |
| hpa..... | 355 |
| hs..... | 361 |
| ht..... | 360 |
| hts..... | 360 |
| hu..... | 364 |
| hz..... | 365 |
| ich..... | 357 |
| ich1..... | 357 |
| idcok()..... | 118 |
| idl原因ok..... | 62 |
| idl原因ok()..... | 119 |
| if..... | 360 |
| il..... | 356 |
| il1..... | 356 |
| immedok()..... | 120 |
| implementation-defined..... | 4 |
| inch()..... | 123 |
| inchnstr()..... | 124 |
| inchstr..... | 124 |
| ind..... | 353, 356 |
| delays..... | 361 |
| independence of print modes assumed..... | 372 |
| indn..... | 353, 356 |
| infocmp..... | 324 |
| initc..... | 363 |
| initialization..... | 27, 360 |
| initialization string..... | 27 |
| initialize a color-pair..... | 363 |
| initp..... | 363 |
| initscr()..... | 126 |
| init_color..... | 56 |
| init_color()..... | 125 |
| init_pair..... | 56, 125 |
| innstr()..... | 128 |
| innwstr()..... | 130 |
| insch()..... | 134 |

| | |
|--------------------------------|-----|
| insdelln() | 135 |
| insert | |
| delay per line | 350 |
| effect on straddling character | 20 |
| resulting rendition | 21 |
| insert/delete character | 357 |
| insert/delete line | 356 |
| insertion | 19 |
| insertln() | 136 |
| insnstr() | 137 |
| insstr | 137 |
| instr | 128 |
| instr() | 138 |
| ins_nwstr() | 132 |
| ins_wch() | 133 |
| ins_wstr | 132 |
| interfaces | |
| implementation | 9 |
| system | 305 |
| use | 9 |
| internationalization | 2 |
| intrflush() | 139 |
| invis | 358 |
| invisible text | 358 |
| inwstr | 130 |
| inwstr() | 140 |
| in_wch() | 121 |
| in_wchnstr() | 122 |
| in_wchstr | 122 |
| ip | 357 |
| iprog | 360 |
| is1, is2, is3 | 360 |
| isendwin() | 143 |
| ISO/IEC 6429: 1992 | 338 |
| is_linetouched() | 141 |
| is_wintouched | 141 |
| it | 360 |
| italic | 372 |
| ka1, ka3 | 360 |
| kb2 | 360 |
| kbs | 360 |
| kc1, kc3 | 360 |
| kclr | 360 |
| kctab | 360 |
| kcub1 | 360 |
| kcud1 | 360 |
| kcuf1 | 360 |
| kcuu1 | 360 |
| kdch1 | 360 |
| kdll | 360 |
| ked | 360 |
| kel | 360 |

| | |
|--------------------------------------|---------|
| keyname() | 144 |
| keypad | 360 |
| keypad() | 145 |
| key_ prefix | 338 |
| key_name | 144 |
| kf0, kf1, and so on | 360 |
| khome | 360 |
| khts | 360 |
| kich1 | 360 |
| kill | 360 |
| kill character | 379 |
| killchar | 89 |
| killchar() | 146 |
| killwchar | 89, 146 |
| kind | 360 |
| kl | 360 |
| km | 365 |
| knp | 360 |
| kpp | 360 |
| kri | 360 |
| krmir | 360 |
| ktbc | 360 |
| last entry in terminfo | 349 |
| LC_CTYPE | 15 |
| Lear Siegler ADM-3 (example) | 354 |
| leaveok | 62 |
| leaveok() | 147 |
| leavok() | 148 |
| left and top edge | 353 |
| left margin | 371 |
| left-to-right writing | 3 |
| legacy | 4 |
| length of line, effect on print rate | 376 |
| letter-quality | 376 |
| lexical conventions | 339 |
| lf | 353 |
| lf0, lf1, and so on | 360 |
| lh | 360 |
| line drawing character | 14 |
| line feed | 371 |
| line graphics | 361 |
| line-drawing macros | 308 |
| line/column coordinate | 18 |
| LINES | 32 |
| lines | 353 |
| LINES | 377 |
| lines on screen | 353 |
| ll | 355 |
| lm | 365 |
| locale | 1 |
| locale-specific | 15 |
| long name of device | 337 |

| | |
|----------------------------------|----------|
| longname() | 149 |
| lpi | 375 |
| recalculate resolution after | 368 |
| lpix | 375 |
| lpi[x] | 368 |
| LSI ADM-3a (example) | 355 |
| lw | 360 |
| macros | |
| line-drawing | 308 |
| maddr | 369 |
| magic cookie glitch | 358 |
| mandatory delay | 350 |
| manipulation of window | 14 |
| margin | 353, 371 |
| may | 4 |
| mc0, mc4, and so on | 365 |
| mcs | 367-368 |
| mcub[1] | 369 |
| mcud[1] | 369 |
| mcuf[1] | 369 |
| mcuu[1] | 369 |
| mcu[b/d/f/u][1] | 370 |
| media copy string | 365 |
| meta key | 365 |
| meta() | 150 |
| mgc | 361, 372 |
| mhpa | 369 |
| reverse motion should not affect | 370 |
| Micro-Term ACT-IV (example) | 355 |
| Micro-Term MIME (example) | 358 |
| mir | 357 |
| mjump | 369 |
| mls | 367 |
| modification outside subwindow | 20 |
| motion, automatic | 372 |
| move() | 151 |
| mrcup | 354 |
| msgr | 358 |
| enhanced printing | 372 |
| multi-byte character | |
| function naming | 25 |
| multi-column character | 16 |
| multiple character functions | 25 |
| must | 4 |
| mv | 152 |
| mv prefix | 25 |
| position arguments | 18 |
| mvaddch | 36 |
| mvaddch() | 156 |
| mvaddchnstr | 37, 157 |
| mvaddchstr | 37 |
| mvaddchstr() | 157 |

| | |
|----------------------|------------|
| mvaddnstr..... | 38 |
| mvaddnstr()..... | 158 |
| mvaddnwstr..... | 40 |
| mvaddnwstr()..... | 159 |
| mvaddstr..... | 38, 158 |
| mvaddwstr..... | 40, 159 |
| mvadd_wch..... | 33 |
| mvadd_wch()..... | 154 |
| mvadd_wchnstr..... | 34 |
| mvadd_wchnstr()..... | 155 |
| mvadd_wchstr..... | 34, 155 |
| mvchgat..... | 60 |
| mvchgat()..... | 160 |
| mvcur()..... | 161 |
| mvdelch..... | 76 |
| mvdelch()..... | 162 |
| mvderwin()..... | 163 |
| mvgetch..... | 102 |
| mvgetch()..... | 165 |
| mvgetnstr..... | 107 |
| mvgetnstr()..... | 167 |
| mvgetn_wstr..... | 105 |
| mvgetn_wstr()..... | 166 |
| mvgetstr..... | 107, 167 |
| mvget_wch..... | 94 |
| mvget_wch()..... | 164 |
| mvget_wstr..... | 105, 166 |
| mvhline..... | 116 |
| mvhline()..... | 168 |
| mvhline_set..... | 117 |
| mvhline_set()..... | 169 |
| mvinch..... | 123 |
| mvinch()..... | 172 |
| mvinchnstr..... | 124 |
| mvinchnstr()..... | 173 |
| mvinchstr..... | 124, 173 |
| mvinnstr..... | 128 |
| mvinnstr()..... | 174 |
| mvinnwstr..... | 130 |
| mvinnwstr()..... | 175 |
| mvinsch..... | 134 |
| mvinsch()..... | 178 |
| mvinsnstr..... | 137 |
| mvinsnstr()..... | 179 |
| mvinsstr..... | 137, 179 |
| mvinstr..... | 128, 174 |
| mvins_nwstr..... | 132 |
| mvins_nwstr()..... | 176 |
| mvins_wch..... | 133 |
| mvins_wch()..... | 177 |
| mvins_wstr..... | 132, 176 |
| mvinwstr..... | 130, 175 |

| | |
|---------------------------------------|------------|
| mvn_wch..... | 121 |
| mvn_wch()..... | 170 |
| mvn_wchnstr..... | 122 |
| mvn_wchnstr()..... | 171 |
| mvn_wchstr..... | 122, 171 |
| mvpa..... | 369 |
| reverse motion should not affect..... | 370 |
| mvprintw()..... | 180 |
| mvscanw()..... | 181 |
| mvvline..... | 116, 168 |
| mvvline_set..... | 117, 169 |
| mvw prefix..... | 25 |
| mvwaddch..... | 36, 156 |
| mvwaddchnstr..... | 37, 157 |
| mvwaddchstr..... | 37, 157 |
| mvwaddnstr..... | 38, 158 |
| mvwaddnwstr..... | 40, 159 |
| mvwaddstr..... | 38, 158 |
| mvwaddwstr..... | 40, 159 |
| mvwadd_wch..... | 33, 154 |
| mvwadd_wchnstr..... | 34, 155 |
| mvwadd_wchstr..... | 34, 155 |
| mvwchgat..... | 60, 160 |
| mvwdelch..... | 76, 162 |
| mvwgetch..... | 102, 165 |
| mvwgetnstr..... | 107, 167 |
| mvwgetn_wstr..... | 105, 166 |
| mvwgetstr..... | 107, 167 |
| mvwget_wch..... | 94, 164 |
| mvwget_wstr..... | 105, 166 |
| mvwhline..... | 116, 168 |
| mvwhline_set..... | 117, 169 |
| mvwin()..... | 182 |
| mvwinch..... | 123, 172 |
| mvwinchnstr..... | 124, 173 |
| mvwinchstr..... | 124, 173 |
| mvwinnstr..... | 128, 174 |
| mvwinnwstr..... | 130, 175 |
| mvwinsch..... | 134, 178 |
| mvwinsnstr..... | 137, 179 |
| mvwinsstr..... | 137, 179 |
| mvwinstr..... | 128, 174 |
| mvwins_nwstr..... | 132, 176 |
| mvwins_wch..... | 133, 177 |
| mvwins_wstr..... | 132, 176 |
| mvwinwstr..... | 130, 175 |
| mvwin_wch..... | 121, 170 |
| mvwin_wchnstr..... | 122, 171 |
| mvwin_wchstr..... | 122, 171 |
| mvwprintw..... | 180 |
| mvwscanw..... | 181 |
| mvwvline..... | 116, 168 |

| | |
|---------------------------------------|------------|
| mvwvline_set..... | 117, 169 |
| n infix | 25 |
| name of capability | 338 |
| name of device..... | 377 |
| name space | |
| X/Open..... | 10 |
| naming | 24 |
| napms() | 183 |
| ncv | 363-364 |
| near-letter-quality..... | 376 |
| nel | 353 |
| network terminal..... | 364 |
| networked asynchronous terminal..... | 27 |
| newline | 353 |
| special processing..... | 21 |
| newpad()..... | 184 |
| newterm..... | 126 |
| newterm() | 186 |
| newwin | 80 |
| newwin()..... | 187 |
| nl()..... | 188 |
| nlab..... | 360 |
| no | 189 |
| nocbreak | 59 |
| nocbreak()..... | 190 |
| nodelay()..... | 191 |
| noecho..... | 84 |
| noecho()..... | 192 |
| non-spacing character..... | 16 |
| non-spacing characters..... | 3 |
| non-standard terminal..... | 27 |
| nonl..... | 188 |
| nonl() | 193 |
| noqiflush() | 194 |
| noraw | 59, 190 |
| notimeout mode | 22 |
| notimeout() | 195 |
| npc | 364 |
| npins..... | 374 |
| nrrmc..... | 356 |
| null chtype..... | 379 |
| null wide-character code..... | 379 |
| numeric capability | 338 |
| OB..... | 5 |
| oc..... | 363 |
| octal specification in terminfo | 350 |
| op | 363 |
| optimization..... | 1 |
| orc | 367 |
| implied change to..... | 368 |
| orhi | 367 |
| implied change to..... | 368 |

| | |
|--|--------------|
| origin..... | 18 |
| orl..... | 367 |
| implied change to..... | 368 |
| orphaned character..... | 18 |
| orphaned column..... | 18 |
| orvi..... | 367 |
| implied change to..... | 368 |
| os..... | 353, 358 |
| overlapping..... | 20 |
| overlay()..... | 196 |
| overstrike..... | 353 |
| overwrite..... | 196 |
| overwriting..... | 18, 20 |
| p prefix..... | 24-25 |
| pad..... | 15, 364, 379 |
| functions that use..... | 25 |
| pad character..... | 364 |
| padding..... | 338 |
| padding character..... | 350 |
| page eject..... | 364 |
| pairs..... | 363 |
| pair_content..... | 56 |
| pair_content()..... | 197 |
| PAIR_NUMBER..... | 197 |
| parametrized string..... | 354 |
| parent window..... | 14, 379 |
| patch..... | 364 |
| pb..... | 361 |
| PC terminal emulator..... | 363 |
| pechochar()..... | 198 |
| pecho_wchar..... | 198 |
| period in terminfo..... | 351 |
| Perkin-Elmer Owl (example)..... | 357 |
| pfkey..... | 360 |
| pfloc..... | 360 |
| px..... | 360 |
| pln..... | 360 |
| pnoutrefresh..... | 184 |
| pnoutrefresh()..... | 199 |
| pop-up window..... | 20 |
| porder..... | 374 |
| position | |
| current or specified..... | 25 |
| postfix..... | 354 |
| prefix on function/argument..... | 24 |
| prefresh..... | 184, 199 |
| print quality..... | 376 |
| printer resolution..... | 366 |
| printer specification in terminfo..... | 366 |
| printing rate..... | 376 |
| printw..... | 180 |
| printw()..... | 200 |

| | |
|---|----------------------|
| property | |
| background | 18 |
| rendition | 18 |
| window | 17 |
| proportional delay | 350 |
| proportional printing..... | 366 |
| prot | 358 |
| protected text | 358 |
| protocol (xon/xoff) | 353 |
| putp()..... | 201 |
| putwin | 111 |
| putwin() | 202 |
| qiflush | 194 |
| qiflush()..... | 203 |
| quality of printing | 376 |
| raster graphics | 374 |
| raw | 59 |
| raw() | 204 |
| rbim | 374 |
| rc..... | 356, 361 |
| inclusion in tsl/fsl | 361 |
| rcsd | 373 |
| reading subwindow | |
| effect on straddling character | 20 |
| redrawwin()..... | 205 |
| reference pages | 29 |
| format..... | 6 |
| refresh | 82 |
| clears touched flag | 14 |
| refresh() | 206 |
| relocation of cursor | 19 |
| rendition | 16 , 359, 379 |
| background | 18 |
| window | 18 |
| rendition of character placed in window | 21 |
| rep..... | 364 |
| replacing characters | 18 |
| resetty() | 208 |
| reset_prog_mode..... | 72 |
| reset_prog_mode()..... | 207 |
| reset_shell_mode..... | 72, 207 |
| resolution..... | 366 |
| resolution, effect of changing | 375 |
| restartterm..... | 73 |
| restartterm()..... | 209 |
| restoring subwindow..... | 20 |
| rev | 358 |
| reverse Polish..... | 354 |
| reverse-video screen | 358 |
| rf..... | 360 |
| rfi..... | 364 |
| ri..... | 353, 356 |

| | |
|--------------------------------------|-----------------|
| right margin | 371 |
| right-to-left writing | 3 |
| rin..... | 353, 356 |
| riponline()..... | 210 |
| ritm | 372 |
| rlm | 370 |
| rmacs | 358 |
| rmcup | 356 |
| rmdc | 357 |
| rmicm | 370 |
| rmir..... | 357 |
| rmkx | 360 |
| rmln..... | 360 |
| rmm..... | 365 |
| rmp..... | 357 |
| rms0..... | 358 |
| rmul..... | 358 |
| rmxon..... | 365 |
| rounding..... | 366 |
| row or column cursor addressing..... | 355 |
| RPN | 354 |
| rs1, rs2 | 360 |
| rshm | 372 |
| rsubm | 372 |
| rsupm | 372 |
| rum | 370 |
| rwidm..... | 372 |
| sam | 370 |
| savetty | 208 |
| savetty()..... | 211 |
| sbim..... | 374 |
| sc..... | 356, 361 |
| inclusion in tsl/fsl | 361 |
| scanw | 181 |
| scanw() | 212 |
| scp..... | 363 |
| screen | 14 , 353 |
| SCREEN..... | 379 |
| screen blink | 358 |
| sclr() | 215 |
| scroll | 215 |
| effect on straddling character | 20 |
| scrolling | 353 |
| scrolling region..... | 356 |
| scrollok..... | 62 |
| scrollok() | 216 |
| scr_dump() | 213 |
| scr_init..... | 213 |
| scr_restore | 213 |
| scr_set..... | 213 |
| scs | 373 |
| scsd..... | 373 |

| | |
|--|------------|
| sdrfq | 376 |
| search path for TERM..... | 377 |
| setab | 363 |
| setaf | 363 |
| setb | 363 |
| setcchar()..... | 219 |
| setf | 363 |
| setscrreg..... | 62 |
| setscrreg()..... | 220 |
| settable scrolling region..... | 356 |
| setupterm..... | 73 |
| setupterm() | 221 |
| set_curterm..... | 73 |
| set_curterm() | 217 |
| set_term() | 218 |
| sgr | 359 |
| sgr0..... | 358 |
| shaded text..... | 5 |
| shadow..... | 372 |
| shadowing..... | 372 |
| shall | 4 |
| sharing definition in terminfo | 366 |
| should | 4 |
| signals..... | 13-14 |
| similar terminal | 366 |
| single-byte character function naming | 25 |
| sitm..... | 372 |
| slash in terminfo..... | 350 |
| slk_atroff() | 222 |
| slk_attron..... | 222 |
| slk_attrset | 222 |
| slk_attr_off..... | 222 |
| slk_attr_on..... | 222 |
| slk_attr_set | 222 |
| slk_clear | 222 |
| slk_color..... | 222 |
| slk_init..... | 222 |
| slk_label..... | 222 |
| slk_noutrefresh | 222 |
| slk_refresh | 222 |
| slk_restore | 222 |
| slk_set..... | 222 |
| slk_touch | 222 |
| slk_wset | 222 |
| slm | 370 |
| smacs..... | 358 |
| smb[b/l/r/t]..... | 371 |
| smcup..... | 356 |
| smdc | 357 |
| smg[b/l/r/t]p..... | 371 |

| | |
|-----------------------------------|---------|
| smicm | 370 |
| smir | 357 |
| smkx | 360 |
| smln | 360 |
| smm | 365 |
| sms0 | 358 |
| smul | 358 |
| smxon | 365 |
| snlq | 376 |
| snrmq | 376 |
| space | |
| use in terminfo | 337 |
| space character | |
| resulting rendition | 21 |
| spacing complex character | 17 |
| spacing of characters | 366 |
| special characters | 20 |
| special keys | 360 |
| special mode | 356 |
| special mode of device | 377 |
| speed of printing | 376 |
| spinh | 374 |
| spinv | 374 |
| sshm | 372 |
| ssubm | 372 |
| ssupm | 372 |
| stack in terminfo | 354 |
| standend() | 224 |
| standout | 224 |
| standout mode | 358 |
| start_color | 56 |
| start_color() | 225 |
| status line | 361 |
| stdscr | 14, 226 |
| straddling character | 20 |
| string capability | 338 |
| string, parametrized | 354 |
| subcs | 372 |
| subpad | 15, 184 |
| subpad() | 227 |
| subscript | 372 |
| characters available | 372 |
| subwin | 80 |
| subwin() | 228 |
| overview | 14 |
| subwindow | 14, 379 |
| character straddling border | 20 |
| sum | 370 |
| supcs | 372 |
| superscript | 372 |
| characters available | 372 |
| swidm | 372 |

| | |
|-----------------------------------|------------|
| switch | 364 |
| synchronous terminal | 27 |
| syncok() | 229 |
| system interfaces | 305 |
| tab | 360 |
| delays | 361 |
| expansion | 355 |
| special processing..... | 21 |
| use in terminfo..... | 337 |
| tab stop | 27 |
| tbc | 360 |
| Tektronix | |
| model of color specification..... | 363 |
| Tektronix 4025 | |
| command character..... | 364 |
| Tektronix 4025 (example)..... | 355 |
| Teleray | |
| destructive tab | 365 |
| Teleray 1061 (example) | 358 |
| termattrs()..... | 230 |
| terminal..... | 15 |
| terminal emulator | 363 |
| terminal-independence | 1, 337 |
| terminfo | 337 |
| TERMINFO | 377 |
| terminfo | |
| format..... | 337 |
| terminology..... | 4 |
| termname() | 231 |
| thread-safety | 13 |
| throughput | 376 |
| tic | 328 |
| tigetflag()..... | 232 |
| tigetnum | 232 |
| tigetstr | 232 |
| tilde, inability to display | 365 |
| timeout..... | 195 |
| timeout() | 235 |
| tiparm..... | 232 |
| tiparm() | 236 |
| top and left edge..... | 353 |
| touch..... | 379 |
| touched | 14 |
| touchline | 141 |
| touchline()..... | 237 |
| touchwin..... | 141, 237 |
| tparam..... | 232 |
| tparam() | 238 |
| tput | 330 |
| tputs() | 239 |
| truncation | 19 |
| tsl | 361 |

| | |
|---|------------|
| TVI 912 (example) | 358 |
| typeahead() | 240 |
| uc | 358 |
| ul | 358 |
| unctrl() | 241 |
| undefined | 5 |
| underline cursor | 358 |
| underlining | 358 |
| ungetch() | 242 |
| unget_wch | 242 |
| uniqueness of terminfo aliases | 337 |
| unspecified | 5 |
| untic | 335 |
| untouchwin | 141 |
| untouchwin() | 243 |
| update | |
| sets touched flag | 14 |
| use | 366 |
| user preference for use of device | 377 |
| use_env() | 244 |
| utilities | 323 |
| variability in print rate | 376 |
| variable-width font | 366 |
| vertical bar | |
| use in terminfo | 337 |
| vi | |
| use of terminfo | 337 |
| vidattr() | 245 |
| video attribute | 16 |
| video enhancement, turn off | 358 |
| vidputs | 245 |
| vid_attr | 245 |
| vid_puts | 245 |
| virtual terminal | 364 |
| visible bell | 358 |
| vline | 116 |
| vline() | 247 |
| vline_set | 117 |
| vline_set() | 248 |
| vpa | 355 |
| vt | 364 |
| VT100 | |
| delayed line wrap | 365 |
| line drawing | 361 |
| scrolling region | 356 |
| status line | 361 |
| vw_printw() | 249 |
| vw_scanw() | 250 |
| w | 251 |
| w infix | 25 |
| w prefix | 24 |
| waddch | 36 |

| | |
|----------------|---------|
| waddch() | 255 |
| waddchnstr | 37, 256 |
| waddchstr | 37 |
| waddchstr() | 256 |
| waddnstr | 38 |
| waddnstr() | 257 |
| waddnwstr | 40 |
| waddnwstr() | 258 |
| waddstr | 38, 257 |
| waddwstr | 40, 258 |
| wadd_wch | 33 |
| wadd_wch() | 253 |
| wadd_wchnstr | 34 |
| wadd_wchnstr() | 254 |
| wadd_wchstr | 34, 254 |
| wattroff | 43 |
| wattroff() | 260 |
| wattron | 43, 260 |
| wattrset | 43, 260 |
| wattr_get | 41 |
| wattr_get() | 259 |
| wattr_off | 41, 259 |
| wattr_on | 41, 259 |
| wattr_set | 41, 259 |
| wbkgd | 46 |
| wbkgd() | 261 |
| wbkgdset | 46, 261 |
| wbkgrnd | 48 |
| wbkgrnd() | 262 |
| wbkgrndset | 48, 262 |
| wborder | 50 |
| wborder() | 263 |
| wborder_set | 52 |
| wborder_set() | 264 |
| wchgat | 60 |
| wchgat() | 265 |
| wclear | 61 |
| wclear() | 266 |
| wclrtobot | 64 |
| wclrtobot() | 267 |
| wclrtoeol | 65 |
| wclrtoeol() | 268 |
| wcoclор_set() | 269 |
| wcolor_set | 41 |
| wcursyncup | 229 |
| wcursyncup() | 270 |
| wdelch | 76 |
| wdelch() | 271 |
| wdeleteln | 77 |
| wdeleteln() | 272 |
| wechochar | 86 |
| wechochar() | 274 |

| | |
|--|----------------------|
| wecho_wchar | 85 |
| wecho_wchar() | 273 |
| werase | 61, 88, 266 |
| wgetbkgrnd | 48, 262 |
| wgetch | 102 |
| wgetch() | 276 |
| wgetnstr | 107 |
| wgetnstr() | 278 |
| wgetn_wstr | 105 |
| wgetn_wstr() | 277 |
| wgetstr | 107, 278 |
| wget_wch | 94 |
| wget_wch() | 275 |
| wget_wstr | 105, 277 |
| whhline_set() | 280 |
| whline | 116 |
| whline() | 279 |
| whline_set | 117 |
| widcs | 368, 372 |
| wide character | 372 |
| wide characters | 3 |
| wide mode | 377 |
| wide-character code (C language) | 379 |
| wide-character string | 379 |
| width of character, variable | 366 |
| will | 5 |
| winch | 123 |
| winch() | 283 |
| winchnstr | 124 |
| winchnstr() | 284 |
| winchstr | 124, 284 |
| wind | 356 |
| window | 14 , 356, 380 |
| clipping | 14 |
| current or specified | 24 |
| parent | 14 |
| touched flag | 14 |
| window background | 18 |
| window hierarchy | 380 |
| window property | 17 |
| window rendition | 18 |
| winnstr | 128 |
| winnstr() | 285 |
| winnwstr | 130 |
| winnwstr() | 286 |
| winsch | 134 |
| winsch() | 289 |
| winsdelln | 135 |
| winsdelln() | 290 |
| winsertln | 136 |
| winsertln() | 291 |
| winsnstr | 137 |

| | |
|-------------------|----------|
| winsnstr() | 292 |
| winsstr | 137, 292 |
| winstr | 128, 285 |
| wins_nwstr | 132 |
| wins_nwstr() | 287 |
| wins_wch | 133 |
| wins_wch() | 288 |
| wins_wstr | 132, 287 |
| winwstr | 130, 286 |
| win_wch | 121 |
| win_wch() | 281 |
| win_wchnstr | 122 |
| win_wchnstr() | 282 |
| win_wchstr | 122, 282 |
| withdrawn | 2 |
| wmove | 151 |
| wmove() | 293 |
| wnoutrefresh | 82 |
| wnoutrefresh() | 294 |
| wprintw | 180 |
| wprintw() | 295 |
| wrap to next line | 353 |
| wrapping | 19 |
| wredrawln | 205 |
| wredrawln() | 296 |
| wrefresh | 82, 294 |
| wscanw | 181, 297 |
| wscanw() | 297 |
| wscrl | 215 |
| wscrl() | 298 |
| wsetscrreg | 62 |
| wsetscrreg() | 299 |
| wsl | 361 |
| wstandend | 224 |
| wstandend() | 300 |
| wstandout | 224, 300 |
| wsyncdown | 229 |
| wsyncdown() | 301 |
| wsyncup | 229, 301 |
| wtimeout | 195 |
| wtimeout() | 302 |
| wtouchln | 141 |
| wtouchln() | 303 |
| wunctrl() | 304 |
| wvline | 116, 279 |
| wvline_set | 117, 280 |
| X/Open name space | 10 |
| xenl | 365 |
| xhp | 365 |
| xhpa | 369 |
| xmc | 358 |
| xoffc | 365 |

| | |
|---------------------------------------|-----------|
| xon | 353, 365 |
| and padding characters | 350 |
| xonc | 365 |
| xsb | 366 |
| xt | 365 |
| xvpa | 369 |
| y, x pair | 18 |
| zero-based row/column numbering | 354 |
| zero-width character | 16 |
| zerom | 371 |

