

z/OS
2.5

C Curses



Note

Before using this information and the product it supports, read the information in [“Notices” on page 179](#).

This edition applies to Version 2 Release 5 of z/OS® (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2021-09-30

© **Copyright International Business Machines Corporation 1996, 2021.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book.....	xxv
Typographical conventions.....	xxv
Other documents.....	xxvi
Where to find more information.....	xxvi
Summary of Changes.....	xxvii
Chapter 1. The Curses Library.....	1
Terminology.....	1
Naming Conventions.....	2
Structure of a Curses Program.....	3
Return Values.....	3
Chapter 2. Initializing Curses.....	5
Chapter 3. Windows in the Curses Environment.....	7
The Default Window Structure.....	7
The Current Window Structure.....	7
Subwindows.....	8
Pads.....	8
Chapter 4. Manipulating Window Data with Curses.....	9
Chapter 5. Curses Interfaces.....	11
addch().....	11
Name.....	11
Synopsis.....	11
Description.....	11
Return Value.....	11
Errors.....	11
Application Usage.....	11
See Also.....	11
addchstr().....	11
Name.....	11
Synopsis.....	12
Description.....	12
Return Value.....	12
Errors.....	12
Application Usage.....	12
See Also.....	12
addnstr().....	12
Name.....	12
Synopsis.....	12
Description.....	13
Return Value.....	13
Errors.....	13
See Also.....	13
addnwstr().....	13
Name.....	13

Synopsis.....	13
Description.....	14
Return Value.....	14
Errors.....	14
See Also.....	14
add_wch().....	14
Name.....	14
Synopsis.....	14
Description.....	14
Return Value.....	15
Errors.....	15
See Also.....	15
add_wchnstr().....	15
Name.....	15
Synopsis.....	15
Description.....	15
Return Value.....	15
Errors.....	15
See Also.....	16
attroff().....	16
Name.....	16
Synopsis.....	16
Description.....	16
Return Value.....	16
Errors.....	16
See Also.....	16
attr_get().....	16
Name.....	16
Synopsis.....	17
Description.....	17
Return Value.....	17
Errors.....	17
See Also.....	17
baudrate().....	18
Name.....	18
Synopsis.....	18
Description.....	18
Return Value.....	18
Errors.....	18
See Also.....	18
beep().....	18
Name.....	18
Synopsis.....	18
Description.....	18
Return Value.....	18
Errors.....	18
Application Usage.....	18
See Also.....	19
bkgd().....	19
Name.....	19
Synopsis.....	19
Description.....	19
Return Value.....	19
bkgd().....	19
Errors.....	19
Application Usage.....	19
See Also.....	20
bkgrnd().....	20

Name.....	20
Synopsis.....	20
Description.....	20
Return Value.....	20
Errors.....	20
See Also.....	20
border().....	21
Name.....	21
Synopsis.....	21
Description.....	21
Return Value.....	21
Errors.....	21
Application Usage.....	22
See Also.....	22
border_set().....	22
Name.....	22
Synopsis.....	22
Description.....	22
Return Value.....	23
Errors.....	23
See Also.....	23
box().....	23
Name.....	23
Synopsis.....	23
Description.....	23
Return Value.....	23
Errors.....	23
Application Usage.....	23
See Also.....	23
box_set().....	23
Name.....	23
Synopsis.....	24
Description.....	24
Return Value.....	24
Errors.....	24
See Also.....	24
can_change_color().....	24
Name.....	24
Synopsis.....	24
Description.....	25
Return Value.....	25
Errors.....	26
Application Usage.....	26
See Also.....	26
cbreak().....	26
Name.....	26
Synopsis.....	26
Description.....	26
Return Value.....	26
Errors.....	26
Application Usage.....	27
See Also.....	27
chgat().....	27
Name.....	27
Synopsis.....	27
Description.....	27
Return Value.....	27
Errors.....	27

See Also.....	27
clear().....	27
Name.....	27
Synopsis.....	28
Description.....	28
Return Value.....	28
Errors.....	28
See Also.....	28
clearok().....	28
Name.....	28
Synopsis.....	28
Description.....	28
Return Value.....	29
Errors.....	29
Application Usage.....	29
See Also.....	29
clrtoebot().....	29
Name.....	29
Synopsis.....	29
Description.....	29
Return Value.....	30
Errors.....	30
See Also.....	30
clrtoeol().....	30
Name.....	30
Synopsis.....	30
Description.....	30
Return Value.....	30
Errors.....	30
See Also.....	30
color_content().....	30
Name.....	30
Synopsis.....	30
Description.....	30
COLOR_PAIRS.....	31
Name.....	31
Synopsis.....	31
Description.....	31
COLS.....	31
Name.....	31
Synopsis.....	31
Description.....	31
See Also.....	31
copywin().....	31
Name.....	31
Synopsis.....	31
Description.....	31
Return Value.....	32
Errors.....	32
See Also.....	32
curscr.....	32
Name.....	32
Synopsis.....	32
Description.....	32
See Also.....	32
curs_set().....	32
Name.....	32
Synopsis.....	32

Description.....	32
Return Value.....	33
Errors.....	33
See Also.....	33
cur_term().....	33
Name.....	33
Synopsis.....	33
Description.....	33
See Also.....	33
def_prog_mode().....	33
Name.....	33
Synopsis.....	33
Description.....	34
Return Value.....	34
Errors.....	34
Application Usage.....	34
See Also.....	34
delay_output().....	34
Name.....	34
Synopsis.....	34
Description.....	34
Return Value.....	34
Errors.....	34
Application Usage.....	35
See Also.....	35
delch().....	35
Name.....	35
Synopsis.....	35
Description.....	35
Return Value.....	35
Errors.....	35
See Also.....	35
del_curterm().....	35
Name.....	35
Synopsis.....	35
Description.....	36
Return Value.....	36
Errors.....	36
Application Usage.....	36
See Also.....	37
deleteln().....	37
Name.....	37
Synopsis.....	37
Description.....	37
Return Value.....	37
Errors.....	37
See Also.....	37
delscreen().....	37
Name.....	37
Synopsis.....	37
Description.....	37
Return Value.....	37
Errors.....	38
See Also.....	38
delwin().....	38
Name.....	38
Synopsis.....	38
Description.....	38

Return Value.....	38
Errors.....	38
See Also.....	38
derwin().....	38
Name.....	38
Synopsis.....	38
Description.....	38
Return Value.....	39
Errors.....	39
Application Usage.....	39
See Also.....	39
douupdate().....	39
Name.....	39
Synopsis.....	39
Description.....	39
Return Value.....	40
Errors.....	40
Application Usage.....	40
See Also.....	40
dupwin().....	40
Name.....	40
Synopsis.....	40
Description.....	40
Return Value.....	40
Errors.....	40
See Also.....	40
echo().....	40
Name.....	40
Synopsis.....	41
Description.....	41
Return Value.....	41
Errors.....	41
See Also.....	41
echochar().....	41
Name.....	41
Synopsis.....	41
Description.....	41
Return Value.....	41
Errors.....	41
Application Usage.....	41
See Also.....	42
echo_wchar().....	42
Name.....	42
Synopsis.....	42
Description.....	42
Return Value.....	42
Errors.....	42
See Also.....	42
endwin().....	42
Name.....	42
Synopsis.....	42
Description.....	42
Return Value.....	43
Errors.....	43
Application Usage.....	43
See Also.....	43
erase().....	43
Name.....	43

Synopsis.....	43
Description.....	43
erasechar().....	43
Name.....	43
Synopsis.....	43
Description.....	44
Return Value.....	44
Errors.....	44
Application Usage.....	44
See Also.....	44
filter().....	44
Name.....	44
Synopsis.....	44
Description.....	44
Return Value.....	45
Errors.....	45
See Also.....	45
flash().....	45
Name.....	45
Synopsis.....	45
Description.....	45
Return Value.....	45
Errors.....	45
Application Usage.....	45
See Also.....	45
flushinp().....	45
Name.....	45
Synopsis.....	45
Description.....	46
Return Value.....	46
Errors.....	46
See Also.....	46
getbegyx().....	46
Name.....	46
Synopsis.....	46
Description.....	46
Return Value.....	46
Errors.....	46
Application Usage.....	46
See Also.....	47
getbkgd().....	47
Name.....	47
Synopsis.....	47
Description.....	47
getbkgrnd().....	47
Name.....	47
Synopsis.....	47
Description.....	47
getcchar().....	47
Name.....	47
Synopsis.....	48
Description.....	48
Return Value.....	48
Errors.....	48
Application Usage.....	48
See Also.....	48
getch().....	48
Name.....	48

Synopsis.....	48
Description.....	49
Return Value.....	49
Errors.....	49
Application Usage.....	49
See Also.....	49
getmaxyx().....	49
Name.....	49
Synopsis.....	49
Description.....	49
getnstr().....	50
Name.....	50
Synopsis.....	50
Description.....	50
Return Value.....	50
Errors.....	50
Application Usage.....	50
See Also.....	51
getn_wstr().....	51
Name.....	51
Synopsis.....	51
Description.....	51
Return Value.....	51
Errors.....	51
Application Usage.....	52
See Also.....	52
getparyx().....	52
Name.....	52
Synopsis.....	52
Description.....	52
getstr().....	52
Name.....	52
Synopsis.....	52
Description.....	52
get_wch().....	52
Name.....	52
Synopsis.....	52
Description.....	53
Return Value.....	53
Errors.....	53
Application Usage.....	53
See Also.....	53
getwin().....	53
Name.....	53
Synopsis.....	53
Description.....	54
Return Value.....	54
Errors.....	54
See Also.....	54
get_wstr().....	54
Name.....	54
Synopsis.....	54
Description.....	54
getyx().....	54
Name.....	54
Synopsis.....	54
Description.....	54
halfdelay().....	55

Name.....	55
Synopsis.....	55
Description.....	55
Return Value.....	55
Errors.....	55
Application Usage.....	55
See Also.....	55
has_colors().....	55
Name.....	55
Synopsis.....	55
Description.....	55
has_ic().....	55
Name.....	55
Synopsis.....	56
Description.....	56
Return Value.....	56
Errors.....	56
Application Usage.....	56
See Also.....	56
hline().....	56
Name.....	56
Synopsis.....	56
Description.....	57
Return Value.....	57
Errors.....	57
hline().....	57
Application Usage.....	57
See Also.....	57
hline_set().....	57
Name.....	57
Synopsis.....	57
Description.....	58
Return Value.....	58
Errors.....	58
hline_set().....	58
See Also.....	58
idcok().....	58
Name.....	58
Synopsis.....	58
Description.....	58
Return Value.....	58
Errors.....	58
See Also.....	58
idlok().....	59
Name.....	59
Synopsis.....	59
Description.....	59
immedok().....	59
Name.....	59
Synopsis.....	59
Description.....	59
Return Value.....	59
Errors.....	59
Application Usage.....	59
See Also.....	59
inch().....	59
Name.....	59
Synopsis.....	60

Description.....	60
Return Value.....	60
Errors.....	60
Application Usage.....	60
See Also.....	60
inchnstr().....	60
Name.....	60
Synopsis.....	60
Description.....	61
Return Value.....	61
Errors.....	61
Application Usage.....	61
See Also.....	61
init_color().....	61
Name.....	61
Synopsis.....	61
Description.....	61
initscr().....	61
Name.....	61
Synopsis.....	61
Description.....	62
initscr().....	62
Return Value.....	62
Errors.....	62
Application Usage.....	62
See Also.....	62
innstr().....	62
Name.....	62
Synopsis.....	63
Description.....	63
Return Value.....	63
Errors.....	63
Application Usage.....	63
See Also.....	63
innwstr().....	64
Name.....	64
Synopsis.....	64
Description.....	64
Return Value.....	64
Errors.....	64
Application Usage.....	64
See Also.....	64
insch().....	65
Name.....	65
Synopsis.....	65
Description.....	65
Return Value.....	65
Errors.....	65
Application Usage.....	65
See Also.....	65
insdeln().....	65
Name.....	65
Synopsis.....	65
Description.....	66
Return Value.....	66
Errors.....	66
See Also.....	66
insertln().....	66

Name.....	66
Synopsis.....	66
Description.....	66
Return Value.....	66
Errors.....	66
See Also.....	66
insnstr().....	67
Name.....	67
Synopsis.....	67
Description.....	67
Return Value.....	67
Errors.....	67
Application Usage.....	67
See Also.....	67
ins_nwstr().....	68
Name.....	68
Synopsis.....	68
Description.....	68
Return Value.....	68
Errors.....	68
See Also.....	68
insstr().....	68
Name.....	68
Synopsis.....	69
Description.....	69
instr().....	69
Name.....	69
Synopsis.....	69
Description.....	69
ins_wch().....	69
Name.....	69
Synopsis.....	69
Description.....	69
Return Value.....	69
Errors.....	69
Application Usage.....	70
See Also.....	70
ins_wstr().....	70
Name.....	70
Synopsis.....	70
Description.....	70
intrflush().....	70
Name.....	70
Synopsis.....	70
Description.....	70
Return Value.....	70
Errors.....	70
Application Usage.....	70
See Also.....	71
in_wch().....	71
Name.....	71
Synopsis.....	71
Description.....	71
Return Value.....	71
Errors.....	71
See Also.....	71
in_wchnstr().....	71
Name.....	71

Synopsis.....	71
Description.....	72
Return Value.....	72
Errors.....	72
Application Usage.....	72
See Also.....	72
inwstr().....	72
Name.....	72
Synopsis.....	72
Description.....	72
isendwin().....	72
Name.....	72
Synopsis.....	72
Description.....	73
Return Value.....	73
Errors.....	73
See Also.....	73
is_linetouched().....	73
Name.....	73
Synopsis.....	73
Description.....	73
Return Value.....	73
Errors.....	74
Application Usage.....	74
See Also.....	74
keyname().....	74
Name.....	74
Synopsis.....	74
Description.....	74
Return Value.....	74
Errors.....	74
Application Usage.....	75
See Also.....	75
keypad().....	75
Name.....	75
Synopsis.....	75
Description.....	75
Return Value.....	75
Errors.....	75
See Also.....	75
killchar().....	75
Name.....	75
Synopsis.....	75
Description.....	76
leaveok().....	76
Name.....	76
Synopsis.....	76
Description.....	76
LINES.....	76
Name.....	76
Synopsis.....	76
Description.....	76
See Also.....	76
longname().....	76
Name.....	76
Synopsis.....	76
Description.....	77
Return Value.....	77

Errors.....	77
Application Usage.....	77
See Also.....	77
meta().....	77
Name.....	77
Synopsis.....	77
Description.....	77
Return Value.....	77
Errors.....	77
Application Usage.....	77
See Also.....	78
move().....	78
Name.....	78
Synopsis.....	78
Description.....	78
Return Value.....	78
Errors.....	78
See Also.....	78
mv.....	78
Name.....	78
Description.....	78
See Also.....	80
mvcur().....	80
Name.....	80
Synopsis.....	80
Description.....	80
Return Value.....	80
Errors.....	80
Application Usage.....	80
See Also.....	80
mvderwin().....	81
Name.....	81
Synopsis.....	81
Description.....	81
Return Value.....	81
Errors.....	81
See Also.....	81
mvprintw().....	81
Name.....	81
Synopsis.....	81
Description.....	82
Return Value.....	82
Errors.....	82
See Also.....	82
mvscanw().....	82
Name.....	82
Synopsis.....	82
Description.....	82
Return Value.....	82
Errors.....	82
See Also.....	82
mvwin().....	82
Name.....	82
Synopsis.....	83
Description.....	83
Return Value.....	83
Errors.....	83
Application Usage.....	83

See Also.....	83
napms().....	83
Name.....	83
Synopsis.....	83
Description.....	83
Return Value.....	83
Errors.....	83
Application Usage.....	83
See Also.....	84
newpad().....	84
Name.....	84
Synopsis.....	84
Description.....	84
Return Value.....	84
Errors.....	84
Application Usage.....	84
See Also.....	85
newterm().....	85
Name.....	85
Synopsis.....	85
Description.....	85
newwin().....	85
Name.....	85
Synopsis.....	85
Description.....	85
nl().....	85
Name.....	85
Synopsis.....	85
Description.....	86
Return Value.....	86
Errors.....	86
Application Usage.....	86
See Also.....	86
no.....	86
Name.....	86
Description.....	86
nodelay().....	86
Name.....	86
Synopsis.....	87
Description.....	87
Return Value.....	87
Errors.....	87
See Also.....	87
noqiflush().....	87
Name.....	87
Synopsis.....	87
Description.....	87
Return Value.....	87
Errors.....	87
Application Usage.....	87
See Also.....	88
notimeout().....	88
Name.....	88
Synopsis.....	88
Description.....	88
Return Value.....	88
Errors.....	88
See Also.....	88

overlay()	88
Name	88
Synopsis	89
Description	89
Return Value	89
Errors	89
See Also	89
pair_content()	89
Name	89
Synopsis	89
Description	89
pechochar()	89
Name	89
Synopsis	90
Description	90
Return Value	90
Errors	90
Application Usage	90
See Also	90
pnoutrefresh()	90
Name	90
Synopsis	90
Description	90
printw()	90
Name	90
Synopsis	91
Description	91
putp()	91
Name	91
Synopsis	91
Description	91
Return Value	91
Errors	91
Application Usage	91
See Also	92
putwin()	92
Name	92
Synopsis	92
Description	92
qiflush()	92
Name	92
Synopsis	92
Description	92
raw()	92
Name	92
Synopsis	92
Description	92
redrawwin()	93
Name	93
Synopsis	93
Description	93
Return Value	93
Errors	93
Application Usage	93
See Also	93
refresh()	93
Name	93
Synopsis	93

Description.....	93
reset_prog_mode().....	94
Name.....	94
Synopsis.....	94
Description.....	94
resetty().....	94
Name.....	94
Synopsis.....	94
Description.....	94
Return Value.....	94
Errors.....	94
See Also.....	94
restartterm().....	94
Name.....	94
Synopsis.....	95
Description.....	95
riponline().....	95
Name.....	95
Synopsis.....	95
Description.....	95
Return Value.....	95
Errors.....	95
Application Usage.....	95
See Also.....	95
savetty().....	96
Name.....	96
Synopsis.....	96
Description.....	96
scanw().....	96
Name.....	96
Synopsis.....	96
Description.....	96
scr_dump().....	96
Name.....	96
Synopsis.....	96
Description.....	96
Return Value.....	97
Errors.....	97
Application Usage.....	97
See Also.....	97
scr1().....	97
Name.....	97
Synopsis.....	97
Description.....	97
Return Value.....	97
Errors.....	98
See Also.....	98
scrollok().....	98
Name.....	98
Synopsis.....	98
Description.....	98
setcchar().....	98
Name.....	98
Synopsis.....	98
Description.....	98
Return Value.....	98
Errors.....	98
See Also.....	98

set_curterm()	99
Name	99
Synopsis	99
Description	99
setscrreg()	99
Name	99
Synopsis	99
Description	99
set_term()	99
Name	99
Synopsis	99
Description	99
Return Value	99
Errors	100
Application Usage	100
See Also	100
setupterm()	100
Name	100
Synopsis	100
Description	100
slk_attroff()	100
Name	100
Synopsis	100
Description	101
Return Value	101
Errors	102
Application Usage	102
See Also	102
standend()	102
Name	102
Synopsis	102
Description	102
Return Value	102
Errors	102
See Also	102
start_color()	103
Name	103
Synopsis	103
Description	103
stdscr	103
Name	103
Synopsis	103
Description	103
See Also	103
subpad()	103
Name	103
Synopsis	103
Description	103
subwin()	104
Name	104
Synopsis	104
Description	104
syncok()	104
Name	104
Synopsis	104
Description	104
Return Value	104
Errors	104

Application Usage.....	104
See Also.....	105
termattrs().....	105
Name.....	105
Synopsis.....	105
Description.....	105
Return Value.....	105
Errors.....	105
See Also.....	105
termname().....	105
Name.....	105
Synopsis.....	105
Description.....	105
Return Value.....	106
Errors.....	106
See Also.....	106
tgetent().....	106
Name.....	106
Synopsis.....	106
Description.....	106
Return Value.....	106
Errors.....	106
Application Usage.....	106
See Also.....	107
tigetflag().....	107
Name.....	107
Synopsis.....	107
Description.....	107
Return Value.....	107
Errors.....	107
Application Usage.....	107
See Also.....	108
timeout().....	108
Name.....	108
Synopsis.....	108
Description.....	108
touchline().....	108
Name.....	108
Synopsis.....	108
Description.....	108
tparm().....	108
Name.....	108
Synopsis.....	109
Description.....	109
tputs().....	109
Name.....	109
Synopsis.....	109
Description.....	109
typeahead().....	109
Name.....	109
Synopsis.....	109
Description.....	109
Return Value.....	109
Errors.....	109
See Also.....	110
unctrl().....	110
Name.....	110
Synopsis.....	110

Description.....	110
Return Value.....	110
Errors.....	110
See Also.....	110
ungetch().....	110
Name.....	110
Synopsis.....	110
Description.....	110
Return Value.....	110
Errors.....	111
See Also.....	111
untouchwin().....	111
Name.....	111
Synopsis.....	111
Description.....	111
use_env().....	111
Name.....	111
Synopsis.....	111
Description.....	111
Return Value.....	111
Errors.....	111
See Also.....	111
vidattr().....	112
Name.....	112
Synopsis.....	112
Description.....	112
Return Value.....	112
Errors.....	112
Application Usage.....	112
See Also.....	113
vline().....	113
Name.....	113
Synopsis.....	113
Description.....	113
vline_set().....	113
Name.....	113
Synopsis.....	113
Description.....	113
vwprintw().....	113
Name.....	113
Synopsis.....	113
Description.....	113
Return Value.....	114
Errors.....	114
Application Usage.....	114
See Also.....	114
vw_printw().....	114
Name.....	114
Synopsis.....	114
Description.....	114
Return Value.....	114
Errors.....	114
Application Usage.....	114
See Also.....	114
vwscanw().....	115
Name.....	115
Synopsis.....	115
Description.....	115

Return Value.....	115
Errors.....	115
Application Usage.....	115
See Also.....	115
vw_scanw().....	115
Name.....	115
Synopsis.....	115
Description.....	115
Return Value.....	115
Errors.....	116
Application Usage.....	116
See Also.....	116
w.....	116
Name.....	116
Description.....	116
wunctrl().....	118
Name.....	118
Synopsis.....	118
Description.....	118
Return Value.....	119
Errors.....	119
See Also.....	119

Chapter 6. Headers..... 121

<curseh>.....	121
Name.....	121
Synopsis.....	121
Description.....	121
See Also.....	134
<termh>.....	134
Name.....	134
Synopsis.....	134
Description.....	134
See Also.....	134
<unctrlh>.....	135
Name.....	135
Description.....	135
See Also.....	135

Chapter 7. Terminfo Source Format (ENHANCED CURSES)..... 137

Source File Syntax.....	137
Minimum Guaranteed Limits.....	138
Formal Grammar.....	138
Defined Capabilities.....	139
Sample Entry.....	152
Types of Capabilities in the Sample Entry.....	152
Device Capabilities.....	154
Basic Capabilities.....	154
Parameterized Strings.....	155
Cursor Motions.....	156
Area Clears.....	156
Insert/Delete Line.....	157
Insert/Delete Character.....	157
Highlighting, Underlining, and Visible Bells.....	158
Keypad.....	160
Tabs and Initialization.....	160
Delays.....	161

Status Lines.....	161
Line Graphics.....	162
Color Manipulation.....	163
Miscellaneous.....	165
Special Cases.....	165
Similar Terminals.....	166
Printer Capabilities.....	166
Rounding Values.....	166
Printer Resolution.....	166
Specifying Printer Resolution.....	167
Capabilities that Cause Movement.....	168
Alternate Character Sets.....	172
Dot-Matrix Graphics.....	173
Effect of Changing Printing Resolution.....	174
Print Quality.....	175
Printing Rate and Buffer Size.....	175
Selecting a Terminal.....	175
Application Usage.....	176
Conventions for Device Aliases.....	176
Variations of Terminal Definitions.....	176
Notices.....	179
Terms and conditions for product documentation.....	180
IBM Online Privacy Statement.....	181
Policy for unsupported hardware.....	181
Minimum supported hardware.....	181
Trademarks.....	182
Glossary.....	183
Index.....	185

About This Book

This manual describes the curses interface for application programs using the z/OS C language. Readers are expected to be experienced C language programmers and to be familiar with open systems standards or a UNIX operating system. This book also assumes that readers are somewhat familiar with MVS systems and with the information for MVS and its accompanying products. Readers also should have read *z/OS Introduction and Release Guide* which describes the services and the concepts of z/OS. This manual is organized as follows:

- Chapter 1, “The Curses Library,” on page 1 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 2. 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 5, “Curses Interfaces,” on page 11 defines the Curses functional interfaces.
- Chapter 6, “Headers,” on page 121 defines the contents of headers, which declare constants, macros and data structures that are needed by programs using the services provided by Chapter 7, “Terminfo Source Format (ENHANCED CURSES),” on page 137.
- Chapter 7, “Terminfo Source Format (ENHANCED CURSES),” on page 137 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. The remainder of the chapter discusses the use of these capabilities by the writer of a terminfo entry to describe the characteristics of the terminal in use.
- The glossary contains definitions of terms used in this manual.

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.

Notes:

- 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>.
- The values of errors are defined in <errno.h>.

Other documents

The following documents are referenced in this specification:

- ANSI standard X3.159-1989, Programming Language C.
- ISO 8859-1:1987, Information Processing - 8-bit Single-byte Coded Graphic Character Sets - Part 1: Latin Alphabet No. 1.
- ISO/IEC 646:1991, Information Processing - ISO 7-bit Coded Character Set for Information Interchange.
- ISO/IEC 9899:1990, Programming Languages - C (technically identical to ANSI standard X3.159-1989).
- System V Interface Definition (Spring 1986 - Issue 2).
- System Interface Definitions (1989 - 3rd Edition).
- 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).
- Operating System API Reference, UNIXO SVR4.2 (1992) (ISBN: 0-13-017658-3).

Where to find more information

For an overview of the information associated with z/OS, see [z/OS Information Roadmap](#).

z/OS Basic Skills in IBM Documentation

z/OS Basic Skills in IBM® Documentation is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. IBM Documentation is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, z/OS Basic Skills is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

z/OS Basic Skills in IBM Documentation (www.ibm.com/docs/en/zos-basic-skills?topic=zosbasics/com.ibm.zos.zbasics/homepage.html) is available to all users (no login required).

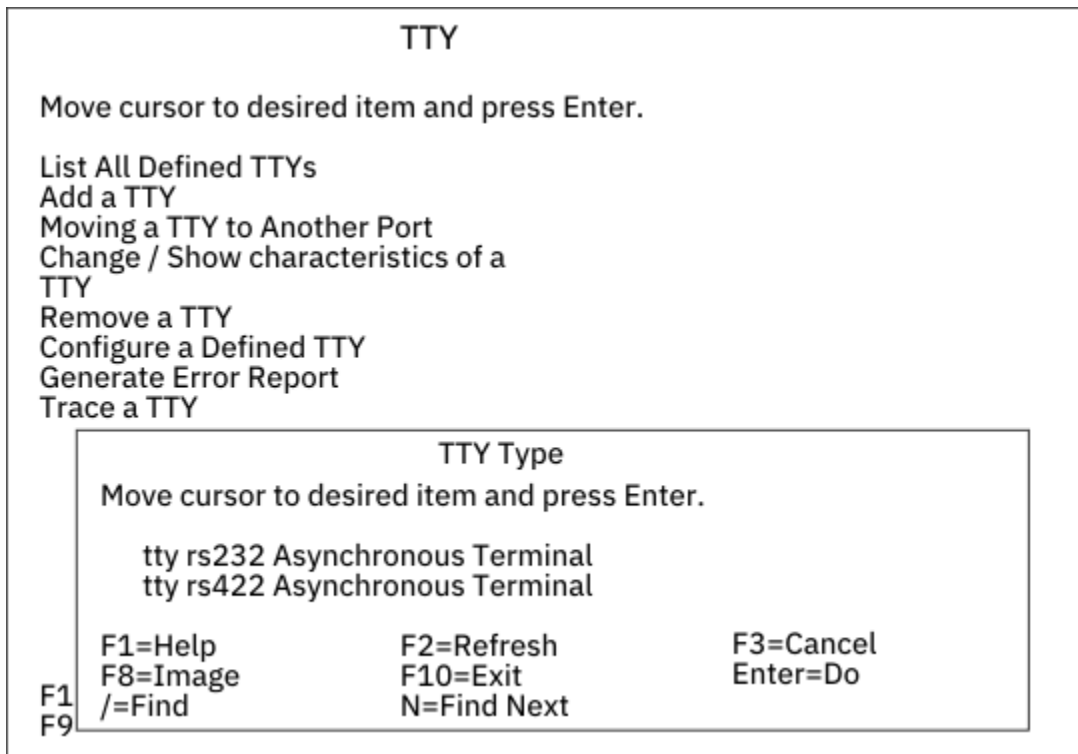
Summary of Changes

**Summary of Changes for
SA38-0690-00
z/OS Version 2 Release 1**

This book contains information also presented in *OS/390 C Curses*, SC28-1907-01.

Chapter 1. The Curses Library

The Curses library provides a set of functions that enable you to manipulate a terminal's display regardless of the terminal type. Throughout this documentation, the Curses library is referred to as `curses`. The basis of `curses` programming is the window data structure. Using this structure, you can manipulate data on a terminal's display. You can instruct `curses` to treat the entire terminal display as one large window or you can create multiple windows on the display. The windows can be different sizes and can overlap one another. The following figure shows a typical `curses` application with a single large window and one subwindow



Each window on a terminal's display has its own window data structure. This structure keeps state information about the window such as its size and where it is located on the display. `Curses` uses the window data structure to obtain relevant information it needs to carry out your instructions.

Terminology

When programming with `curses`, you should be familiar with the following terms:

Term

Definition

current character

The character that the logical cursor is currently on.

current line

The line that the logical cursor is currently on.

curscr

A virtual default window provided by `curses`. The `curscr` (current screen) is an internal representation of what currently appears on the terminal's external display. You should not modify the `curscr`.

display

A physical display connected to a workstation.

logical cursor

The cursor location within each window. The window data structure keeps track of the location of its logical cursor.

pad

A type of window that is larger than the dimensions of the terminal's display. Unlike other windows, a pad is not associated with any particular portion of the display.

physical cursor

The cursor that appears on a display. The workstation uses this cursor to write to the display. There is only one physical cursor per display. To change the position of the physical cursor, you must do a refresh.

screen

The window that fills the entire display. The screen is synonymous with the `stdscr` (standard screen).

stdscr

A virtual default window provided by curses that represents the entire display.

window

A pointer to a C data structure and the graphic representation of that data structure on the display. A window can be thought of as a two-dimensional array representing how all or part of the display looks at any point in time. Windows range in size from the entire display to a single character.

Naming Conventions

A single curses function can have two or more versions. Curses functions with multiple versions follow distinct naming conventions that identify the separate versions. These conventions add a prefix to a standard curses function and identify what arguments the function requires or what actions take place when the function is called. The different versions of curses function names use three prefixes:

Prefix**Description****w**

Identifies a function that requires a window argument.

p

Identifies a function that requires a pad argument.

mv

Identifies a function that first performs a move to the program-supplied coordinates.

Some curses functions with multiple versions do not include one of the preceding prefixes. These functions use the curses default window `stdscr` (standard screen). The majority of functions that use the `stdscr` are functions created in the `/usr/include/curses.h` file using `#define` statements. The preprocessor replaces these statements at compilation time. As a result, these functions do not appear in the compiled assembly code, a trace, a debugger, or the curses source code.

If a curses function has only a single version, it does not necessarily use `stdscr`. For example, the `printw()` function prints a string to the `stdscr`. The `wprintw()` function prints a string to a specific window by supplying the Window argument. The `mvprintw()` function moves the specified coordinates to the `stdscr` and then performs the same function as the `printw()` function. Likewise, the `mvwprintw()` function moves the specified coordinates to the specified window and then performs the same function as the `wprintw()` function.

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 same 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 manual page specifies the outcome if the value of `n` is inappropriate.

Structure of a Curses Program

In general, a curses program has the following progression:

- Start curses.
- Check for color support (optional).
- Start color (optional).
- Create one or more windows.
- Manipulate windows.
- Destroy one or more windows window.
- Stop curses.

Your program does not have to follow this progression exactly.

Return Values

With a few exceptions, all curses functions return either the integer value ERR or the integer value OK. Subroutines that do not follow this convention are noted appropriately. Subroutines that return pointers always return a null pointer on an error.

Chapter 2. Initializing Curses

You must include the **curses.h** file at the beginning of any program that calls curses functions. To do this, use the following statement:

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

Before you can call functions that manipulate windows or screens, you must call the **initscr()** or **newterm()** function. These functions first save the terminal's settings. These functions then call the **setupterm()** function to establish a curses terminal.

Before exiting a curses program, you must call the **endwin()** function. The **endwin()** function restores tty modes, moves the cursor to the lower left corner of the screen, and resets the terminal into the proper nonvisual mode. You can also temporarily suspend curses. If you need to suspend curses, use a shell escape or system call for example. To resume after a temporary escape, you should call the **wrefresh()** or **doupdate()** function. The **isendwin()** function is helpful if, for optimization reasons, you don't want to call the **wrefresh()** function needlessly. You can determine if the **endwin()** function was called without any subsequent calls to the **wrefresh()** function by using the **isendwin()** function.

Most interactive, screen-oriented programs require character-at-a-time input without echoing the result to the screen. To establish your program with character-at-a-time input, call the **cbreak()** and **noecho()** functions after calling the **initscr** function. When accepting this type of input, programs should also call the following functions:

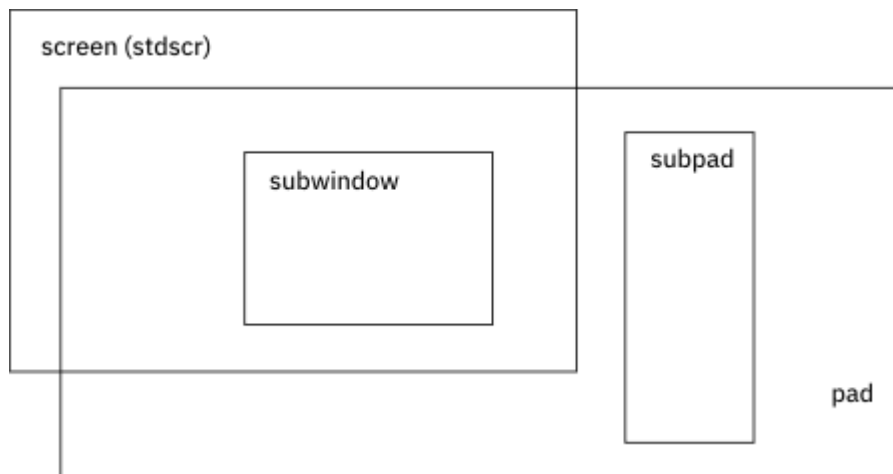
- **nonl()** function.
- **intrflush()** function with the Window parameter set to the **stdscr** and the Flag parameter set to **FALSE**. The Window parameter is required but ignored. You can use **stdscr** as the value of the Window parameter, because **stdscr** is already created for you.
- **keypad()** function with the Window parameter set to the **stdscr** and the Flag parameter set to **TRUE**.

Chapter 3. Windows in the Curses Environment

A curses program manipulates windows that appear on a terminal's display. A window is a rectangular portion of the display. A window can be as large as the entire display or as small as a single character in length and height.

Note: Pads are the exception. A pad is a window that is not restricted by the size of the screen. For more information, see “Pads” on page 8.

The following figure shows the different types of windows that exist in the curses environment:



Within a curses program, windows are variables declared as type `WINDOW`. The `WINDOW` data type is defined in the `/usr/include/curses.h` file as a C data structure. You create a window by allocating a portion of a machine's memory for a window structure. This structure describes the characteristics of the window. When a program changes the window data internally in memory, it must use the **wrefresh()** function (or equivalent function) to update the external, physical screen to reflect the internal change in the appropriate window structure.

Curses supplies a default window when the Curses library is initialized. You can create your own windows known as user-defined windows. Except for the amount of memory available to a program, there is no limit to the number of windows you can create. A curses program can manipulate the default window, user-defined windows, or both.

The Default Window Structure

Curses provides a virtual default window called `stdscr`. The `stdscr` represents, in memory, the entire terminal display. The `stdscr` window structure is created automatically when the Curses library is initialized and it describes the display. When the library is initialized, the length and width variables are set to the length and width of the physical display.

In addition to the `stdscr`, you can define your own windows. These windows are known as user-defined windows to distinguish them from the `stdscr`. Like the `stdscr`, user-defined windows exist in machine memory as structures.

Programs that use the `stdscr` first manipulate the `stdscr` and then call the **refresh()** function to refresh the external display so that it matches the `stdscr` window.

The Current Window Structure

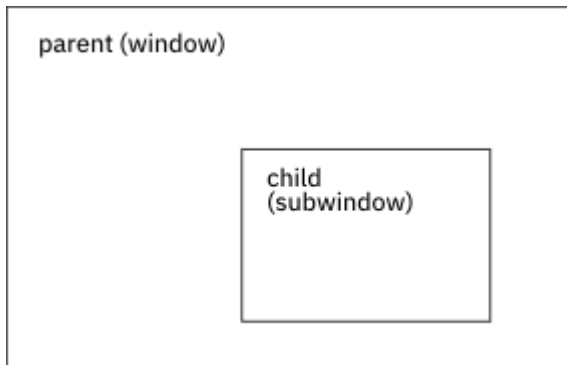
Curses also supports another virtual window called `curscr` (current screen). The `curscr` window is an internal representation of what currently appears on the terminal's external display.

When a program requires the external representation to match the internal representation, it must call a function, such as the **wrefresh()** function, to update the physical display (or the **refresh()** function if the program is working with the stdscr). When a refresh is called on an internal window, curses copies the changed portions of the window into the curscr and updates the physical display.

The curscr is reserved for internal use by curses. You should not manipulate the curscr.

Subwindows

Curses also allows you to construct subwindows. Subwindows are rectangular portions within other windows. A subwindow is also of type WINDOW. The window that contains a subwindow is known as the subwindow's parent and the subwindow is known as the containing window's child. The following figure demonstrates the parent child relationship.



Changes to either the parent window or the child window within the area overlapped by the subwindow are made to both windows. After modifying a subwindow, you should call the **touchline()** or **touchwin()** function on the parent window before refreshing it. The **touchline()** and **touchwin()** functions instruct curses to discard its optimization information for the parent window and to consider the window as having changed. A refresh called on the parent refreshes the children as well.

A subwindow can also be a parent window. The process of layering windows inside of windows is called nesting. The number of nested subwindows is limited to the amount of memory available up to the value of SHRT_MAX as defined in the `/usr/include/limits.h` file. Before you can delete a parent window, you must first delete all of its children using the **delwin()** function. Curses returns an error if you try to delete a window before removing all of its children.

Pads

A pad is a type of window that is not restricted by the terminal's display size or associated with a particular part of the display. You can use pads whenever your program requires a large window. Because a pad is usually larger than the physical display, only a portion of a pad is visible to the user at a given time.

Use pads when you have a large amount of related data that you want to keep all together in one window but you do not need to display all of the data at once.

Windows within pads are known as subpads. Subpads are positioned within a pad at coordinates relative to the parent pad. This placement differs from subwindows which are positioned using screen coordinates.

You should use the **prefresh()** function to show a portion of a pad on the display. Unlike other windows, scrolling or echoing of input does not automatically refresh a pad. Like subwindows, when changing the image of a subpad, you must call either the **touchline()** or **touchwin()** function on the parent pad before refreshing the parent. You can use all the curses function with pads except for the **newwin()**, **subwin()**, **wrefresh()**, and **wnoutrefresh()** functions. These functions are replaced with the **newpad()**, **subpad()**, **prefresh()**, and **pnoutrefresh()** functions.

Chapter 4. Manipulating Window Data with Curses

When curses is initialized, the stdscr is provided automatically. You can manipulate the stdscr using the curses function library or you can create your own, user-defined windows.

Chapter 5. Curses Interfaces

This chapter describes the Curses functions, macros and external variables to support application portability at the C-language source level. The interface definitions are collated as though any underscore characters were not present.

addch()

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 wrapping. These functions perform special-character processing.

Return Value

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

Errors

No errors are defined.

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.

See Also

`add_wch()`, `attroff()`, `doupdate()`, `<curses.h>`.

addchstr()

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

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

Return Value

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

Errors

No errors are defined.

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.

See Also

addch(), *add_wch()*, *add_wchstr()*, **<curses.h>**.

addnstr()

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

```
#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, char *const str, int n);
int mvwaddstr(WINDOW *win, int y, int x, char *const 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.

See Also

addnwstr(), *mbstowcs()*, **<curses.h>**.

addnwstr()

Name

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

Synopsis

```

#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.

See Also

`add_wch()`, `<curses.h>`

add_wch()

Name

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

Synopsis

```
#include <curses.h>

int add_wch(cchar_t *const wch);

int wadd_wch(WINDOW *win, cchar_t *const wch);

int mvadd_wch(int y, int x, cchar_t *const wch);

int mvwadd_wch(WINDOW *win, int y, int x, cchar_t *const 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 wrapping. These functions perform special-character processing.

- 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.

See Also

addch(), `<curses.h>`.

add_wchnstr()

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

```
#include <curses.h>

int add_wchnstr(const cchar_t *wchstr, int n);
int add_wchstr(const cchar_t *wchstr);
int wadd_wchnstr(WINDOW *win, const cchar_t *wchstr, int n);
int wadd_wchstr(WINDOW *win, 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);
```

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.

The 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.

See Also

`<curses.h>`.

attroff()

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.

See Also

attr_get(), *standend()*, `<curses.h>`.

attr_get()

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

```
#include <curses.h>

int attr_get(attr_t *attr, short *color_pair_number, void *opts);
int attr_off(attr_t attr, void *opts);
int attr_on(attr_t attr, void *opts);
int attr_set(attr_t attr, short color_pair_number, void *opts);
int color_set(short color_pair_number, void *opts);
int wattr_get(WINDOW *win, attr_t *attr, short *color_pair_number,
              void *opts);
int wattr_off(WINDOW *win, attr_t attr, void *opts);
int wattr_on(WINDOW *win, attr_t attr, void *opts);
int wattr_set(WINDOW *win, attr_t attr, 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 *attr* 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 *attr* in the current or specified window without affecting any others.

The `attr_on()` and `wattr_on()` functions turn on *attr* 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 *attr* 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

The `attr_get()` and `wattr_get()` functions return the current window attributes for the current or specified window.

The other functions always return OK.

Errors

No errors are defined.

See Also

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

baudrate()

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.

See Also

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

beep()

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.

Application Usage

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

See Also

flash(), `<curses.h>`.

bkgd()

Name

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

Synopsis

```
#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, OR the requested attributes into the window rendition, and set the background attributes 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, 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 background 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, `bkgd()` and `wbkgd()` return OK. Otherwise, they return ERR.

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

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

bkgd()

Errors

No errors are defined.

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.

See Also

<curses.h>.

bkgrnd()

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 complex character

Synopsis

```
#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 background 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.

See Also

<curses.h>.

border()

Name

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

Synopsis

```
#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_BLCORNER
<i>br</i>	Corner of the last line and the ending column	ACS_BRCORNER

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.

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.

See Also

border_set(), *box()*, *hline()*, <curses.h>.

border_set()

Name

border_set, *wborder_set*, - draw borders from complex characters and renditions

Synopsis

```
#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_BLCORNER
<i>br</i>	Corner of the last line and the ending column	WACS_BRCORNER

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.

See Also

box_set(), *hline_set()*, **<curses.h>**.

box()

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, `box()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

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.

See Also

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

box_set()

Name

box_set - draw borders from complex characters and renditions

Synopsis

```
#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, this function returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

See Also

border_set(), *hline_set()*, **<curses.h>**.

[can_change_color\(\)](#)

Name

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

Synopsis

```
#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>`. 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 below.) 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 below.)

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 lower 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`.

Curses

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.

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.

See Also

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

cbreak()

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.

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.

See Also

`<curses.h>`.

chgat()

Name

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

Synopsis

```
#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 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. 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 edition of this document. 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.

See Also

`setcchar()`, `<curses.h>`

clear()

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

The `clear()`, `erase()`, `wclear()` and `werase()` 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.

See Also

clearok(), *doupdate()*, **<curses.h>**.

clearok()

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. 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, `setscrreg()` and `wsetscrreg()` return OK. Otherwise, they return ERR.

The other functions always return OK.

Errors

No errors are defined.

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.

See Also

`clear()`, `delscreen()`, `doupdate()`, `scrll()`, **<curses.h>**

clrrobot()

Name

`clrrobot`, `wclrrobot` - clear from cursor to end of window

Synopsis

```
#include <curses.h>

int clrrobot(void);

int wclrrobot(WINDOW *win);
```

Description

The `clrrobot()` and `wclrrobot()` 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.

Return Value

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

Errors

No errors are defined.

See Also

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

clrtoeol()

Name

clrtoeol, wclrtoeol - clear from cursor to end of line

Synopsis

```
#include <curses.h>

int clrtoeol(void);

int wclrtoeol(WINDOW *win);
```

Description

The *clrtoeol()* and *wclrtoeol()* functions erase the current line from the cursor to the end of the line, inclusive, in the current or specified window.

Return Value

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

Errors

No errors are defined.

See Also

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

color_content()

Name

color_content - identify red/green/blue intensity of a color

Synopsis

```
#include <curses.h>

int color_content(short color, short *red, short *green, short *blue);
```

Description

Refer to *can_change_color()*

COLOR_PAIRS

Name

COLOR_PAIRS, COLORS - external variables for color support

Synopsis

```
#include <curses.h>

extern int COLOR_PAIRS;

extern int COLORS;
```

Description

Refer to *can_change_color()*.

COLS

Name

COLS - number of columns on terminal screen

Synopsis

```
#include <curses.h>

extern int COLS;
```

Description

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

See Also

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

copywin()

Name

copywin - copy a region of a window

Synopsis

```
#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, `copywin()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

See Also

`newpad()`, `overlay()`, `<curses.h>`.

curscr

Name

`curscr` - current window

Synopsis

```
#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.

See Also

`clearok()`, `<curses.h>`.

curs_set()

Name

`curs_set` - set the cursor mode

Synopsis

```
#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 visibility	Appearance of Cursor
0	Invisible
1	Terminal-specific normal mode

Value of visibility	Appearance of Cursor
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 `curs_set()` returns the previous cursor state. Otherwise, the function returns `ERR`.

Errors

No errors are defined.

See Also

`<curses.h>`.

cur_term()

Name

`cur_term` - current terminal information

Synopsis

```
#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.

See Also

`set_curterm()`, `tigetflag()`, `<term.h>`.

def_prog_mode()

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.

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.

See Also

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

delay_output()

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, `delay_output()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

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

See Also

`napms()`, `<curses.h>`.

delch()

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. This function does not change the cursor position.

Return Value

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

Errors

No errors are defined.

See Also

`<curses.h>`.

del_curterm()

Name

`del_curterm`, `restartterm`, `set_curterm`, `setupterm` - interfaces to the terminfo database

Synopsis

```
#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);
```

```
extern TERMINAL *cur_term;
```

Description

These functions retrieve information from the terminfo database.

To gain access to the terminfo database, `setupterm()` 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, `setupterm()` 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, `set_curterm()` 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.

Application Usage

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

See Also

baudrate(), *erasechar()*, *has_ic()*, *longname()*, *putc()*, *termattrs()*, *termname()*, *tgetent()*, *tigetflag()*, *use_env()*, **<term.h>**.

deleteln()

Name

deleteln, *wdeleteln* - delete lines in a window

Synopsis

```
#include <curses.h>

int deleteln(void);

int wdeleteln(WINDOW *win);
```

Description

The *deleteln()* and *wdeleteln()* 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.

See Also

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

delscreen()

Name

delscreen - free storage associated with a screen

Synopsis

```
#include <curses.h>

void delscreen(SCREEN *sp);
```

Description

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

Return Value

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

Errors

No errors are defined.

See Also

endwin(), *initscr()*, **<curses.h>**.

delwin()

Name

delwin - delete a window

Synopsis

```
#include <curses.h>

int delwin(WINDOW *win);
```

Description

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

Return Value

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

Errors

No errors are defined.

See Also

derwin(), *dupwin()*, **<curses.h>**.

derwin()

Name

derwin, newwin, subwin - window creation functions

Synopsis

```
#include <curses.h>

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

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 at (*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 `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.

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);
```

See Also

`delwin()`, `is_linetouched()`, `doupdate()`, **<curses.h>**.

doupdate()

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.

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.

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.

See Also

clearok(), *redrawwin()*, **<curses.h>**.

dupwin()

Name

dupwin - duplicate a window

Synopsis

```
#include <curses.h>

WINDOW *dupwin(WINDOW *win);
```

Description

The `dupwin()` function creates a duplicate of the window *win*.

Return Value

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

Errors

No errors are defined.

See Also

derwin(), *doupdate()*, **<curses.h>**.

echo()

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 for the current screen 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.

See Also

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

echochar()

Name

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

Synopsis

```
#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.

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.

See Also

addch(), *doupdate()*, *echo_wchar()*, **<curses.h>**.

echo_wchar()

Name

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

Synopsis

```
#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.

See Also

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

endwin()

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, `endwin()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

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()`.

See Also

`delscreen()`, `doupdate()`, `initscr()`, `isendwin()`, `<curses.h>`.

erase()

Name

`erase`, `werase` - clear a window

Synopsis

```
#include <curses.h>

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

Description

Refer to `clear()`.

erasechar()

Name

`erasechar`, `erasewchar`, `killchar`, `killwchar` - terminal environment query functions

Synopsis

```
#include <curses.h>

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

Description

The `erasechar()` function returns the current erase character. The `erasewchar()` 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.

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 `killchar()` returns the line kill character. The return value is unspecified when these characters are multi-byte characters.

Upon successful completion, `erasewchar()` and `killwchar()` return OK. Otherwise, they return ERR.

Errors

No errors are defined.

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 `erasewchar()` and `killwchar()` functions overcome these limitations.

See Also

`clearok()`, `delscreen()`, `tcgetattr()`, **<curses.h>**.

filter()

Name

filter - disable use of certain terminal capabilities

Synopsis

```
#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`, `cud`, `cud1`, `cup`, `cuu1` 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.

See Also

`initscr()`, `<curses.h>`.

flash()

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.

Application Usage

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

See Also

`beep()`, `<curses.h>`

flushinp()

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.

See Also

`<curses.h>`.

getbegyx()

Name

`getbegyx`, `getmaxyx`, `getparyx`, `getyx` - get cursor and window coordinates

Synopsis

```
#include <curses.h>

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`.

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`.

Return Value

No return values are defined.

Errors

No errors are defined.

Application Usage

These interfaces are macros and ‘&’ cannot be used before the `y` and `x` arguments. Traditional implementations have often defined the following macros:

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

```
void getmaxx(WINDOW *win, int x);
void getmaxy(WINDOW *win, int y);
void getparx(WINDOW *win, int x);
void getpary(WINDOW *win, int y);
```

Although `getbegyx()`, `getmaxyx()` and `getparyx()` provide the required functionality, this does not preclude applications from defining these macros for their own use. For example, to implement `void getbegx(WINDOW *win, int x);` the macro would be

```
#define getbegx(_win,_x); /
{ /
    int _y; /
    getbegyx(_win,_y,_x); /
}
```

See Also

<curses.h>

getbkgd()

Name

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

Synopsis

```
#include <curses.h>

chtype getbkgd(WINDOW *win);
```

Description

Refer to `bkgd()`.

getbkgrnd()

Name

`getbkgrnd` - get background character and rendition

Synopsis

```
#include <curses.h>

int getbkgrnd(cchar_t *ch);
```

Description

Refer to `bkgrnd()`.

getcchar()

Name

`getcchar` - get a wide character string and rendition from a `cchar_t`

Synopsis

```
#include <curses.h>

int getcchar(const cchar_t *wcval, 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 *wcval*, 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 *wcval* 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 *wcval* and does not change the objects pointed to by *attrs* or *color_pair*.

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

Return Value

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

When *wch* is not a null pointer, `getcchar()` returns OK upon successful completion, and ERR otherwise.

Errors

No errors are defined.

Application Usage

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

See Also

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

getch()

Name

`getch`, `wgetch`, `mvgetch`, `mvwgetch` - 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>**.

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 and then the character at the resulting cursor position is deleted as though <code>delch()</code> were called, except that if the cursor was originally in the first column of the line, then the user is alerted as though <code>beep()</code> were called.
Function keys	The user is alerted as though <code>beep()</code> 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, **`getch()`**, **`mvgetch`**, **`mvwgetch()`** and **`wgetch()`** return the single-byte character, `KEY_` value, or `ERR`. When in the `nodelay` mode and no data is available, `ERR` is returned.

Errors

No errors are defined.

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.

See Also

*`cbreak()`, `doupdate()`, `insch()`, **<curses.h>**.*

getmaxyx()

Name

`getmaxyx` - get size of a window

Synopsis

```
#include <curses.h>

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

Description

Refer to `getbegyx()`.

getnstr()

Name

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

Synopsis

```
#include <curses.h>

int getnstr(char *str, int n);
int getstr(char *str);
int mvgetnstr(int y, int x, char *str, int n);
int mvgetstr(int y, int x, char *str);
int mvwgetnstr(WINDOW *win, int y, int x, char *str, int n);
int mvwgetstr(WINDOW *win, int y, int x, char *str);
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 or carriage return 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 read at most *n* 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()` and then a series of calls to `getch()`. The `mvwgetstr()` function is identical to `getstr()` except it is as though a call to `wmove()` is made and then 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()` and then a series of calls to `getch()`. The `mvwgetnstr()` function is identical to `getnstr()` except it is as though a call to `wmove()` is made and then 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.

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.

See Also

beep(), *getch()*, **<curses.h>**.

getn_wstr()

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

```
#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()* and then a series of calls to *get_wch()* were made. The effect of *mvwget_wstr()* is as though a call to *wmove()* and then a series of calls to *wget_wch()* were made. The effect of *mvget_nwstr()* is as though a call to *move()* and then a series of calls to *get_wch()* were made. The effect of *mvwget_nwstr()* is as though a call to *wmove()* and then 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.

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.

See Also

`get_wch()`, `getstr()`, `<curses.h>`, `<wchar.h>`.

getparyx()

Name

`getparyx` - get subwindow origin coordinates

Synopsis

```
#include <curses.h>

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

Description

Refer to `getbegyx()`.

getstr()

Name

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

Synopsis

```
#include <curses.h>

int getstr(char *str);
```

Description

Refer to `getnstr()`.

get_wch()

Name

`get_wch`, `mvget_wch`, `mvwget_wch`, `wget_wch` - get a wide character from a terminal

Synopsis

```
#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.

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 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.

Application Usage

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

When using these functions, nocbreak mode and echo mode 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.

See Also

beep(), *cbreak()*, *ins_wch()*, *keypad()*, *move()*, **<curses.h>**, **<wchar.h>**.

getwin()

Name

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

Synopsis

```
#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, `getwin()` returns a pointer to the window it created. Otherwise, it returns a null pointer.

Upon successful completion, `putwin()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

See Also

`scr_dump()`, `<curses.h>`.

get_wstr()

Name

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

Synopsis

```
#include <curses.h>

int get_wstr(wint_t *wstr);
```

Description

Refer to `getn_wstr()`.

getyx()

Name

`getyx` - get cursor coordinates

Synopsis

```
#include <curses.h>

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

Description

Refer to `getbegyx()`.

halfdelay()

Name

halfdelay - control input character delay mode

Synopsis

```
#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 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, `halfdelay()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

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

See Also

`cbreak()`, `<curses.h>`.

has_colors()

Name

has_colors - indicate whether terminal supports colors

Synopsis

```
#include <curses.h>

bool has_colors(void);
```

Description

Refer to `can_change_color()`.

has_ic()

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- and delete-character capabilities.

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

Return Value

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

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

Errors

No errors are defined.

Application Usage

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

See Also

`<curses.h>`.

hline()

Name

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

Synopsis

```
#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.

hline()

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.

See Also

`border()`, `box()`, `hline_set()`, `<curses.h>`.

hline_set()

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

```
#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, cchar_t *const 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.

hline_set()

See Also

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

idcok()

Name

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

Synopsis

```
#include <curses.h>

void idcok(WINDOW *win, bool bf);
```

Description

The `idcok()` function specifies whether the implementation may use hardware insert- 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.

See Also

clearok(), *doupdate()*, **<curses.h>**.

idlok()

Name

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

Synopsis

```
#include <curses.h>

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

Description

Refer to `clearok()`.

immedok()

Name

immedok - enable or disable immediate terminal refresh

Synopsis

```
#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.

Application Usage

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

See Also

`clearok()`, `doupdate()`, `<curses.h>`.

inch()

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.

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.

See Also

<curses.h>.

inchnstr()

Name

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

Synopsis

```
#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.

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.

See Also

`inch()`, `<curses.h>`.

init_color()

Name

`init_color`, `init_pair` - redefine specified color or color pair

Synopsis

```
#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()`.

initscr()

Name

`initscr`, `newterm` - screen initialization functions

Synopsis

```
#include <curses.h>

WINDOW *initscr(void);
SCREEN *newterm(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.

initscr()

Return Value

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

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

Errors

No errors are defined.

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()`.

See Also

`delscreen()`, `doupdate()`, `del_curterm()`, `filter()`, `slk_attroff()`, `use_env()`, **<curses.h>**.

innstr()

Name

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

Synopsis

```
#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 `instr()`, `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, `instr()`, `mvinstr()`, `mvwinstr()` and `winstr()` return OK.

Upon successful completion, `innstr()`, `mvinnstr()`, `mvwinnstr()` and `winnstr()` return the number of characters actually read into the string. Otherwise, all these functions return ERR.

Errors

No errors are defined.

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.

See Also

`<curses.h>`.

innwstr()

Name

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

Synopsis

```
#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, `inwstr()`, `mvinwstr()`, `mvwinwstr()` and `winwstr()` return OK.

Upon successful completion, `innwstr()`, `mvinnwstr()`, `mvwinnwstr()` and `winnwstr()` return the number of characters actually read into the string. Otherwise, all these functions return ERR.

Errors

No errors are defined.

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.

See Also

`<curses.h>`.

insch()

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 perform wrapping. 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.

Return Value

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

Errors

No errors are defined.

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.

See Also

ins_wch() <curses.h>.

insdelln()

Name

insdelln, winsdelln - delete or insert lines into a window

Synopsis

```
#include <curses.h>

int insdelln(int n);

int winsdelln(WINDOW *win, int n);
```

Description

The `insdelln()` and `winsdelln()` 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.

See Also

deleteln(), *insertln()*, **<curses.h>**.

insertln()

Name

`insertln`, `winsertln` - insert lines into a window

Synopsis

```
#include <curses.h>

int insertln(void);

int winsertln(WINDOW *win);
```

Description

The `insertln()` and `winsertln()` 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.

See Also

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

insnstr()

Name

insnstr, insstr, mvinsnstr, mvinsstr, mvwinsnstr, mvwinsstr, winsnstr, winsstr - insert a multi-byte character string into a window

Synopsis

```
#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. The `innstr()` and `innwstr()` functions perform wrapping. The `instr()` and `inswstr` 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.

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.

See Also

`<curses.h>`

ins_nwstr()

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

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

The `ins_nwstr()`, `mvins_nwstr()`, `mvwins_nwstr()` and `wins_nwstr()` functions insert at most *n* `wchar_t` characters. If *n* is less than 1, then the entire string is inserted.

Return Value

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

Errors

No errors are defined.

See Also

`<curses.h>`.

insstr()

Name

insstr - insert a multi-byte character string into the current window

Synopsis

```
#include <curses.h>

int insstr(const char *str);
```

Description

Refer to insnstr().

instr()

Name

instr - input a multi-byte character string from the current window

Synopsis

```
#include <curses.h>

int instr(char *str);
```

Description

Refer to innstr().

ins_wch()

Name

ins_wch, mvins_wch, mvwins_wch, wins_wch - insert a complex character and rendition into a window

Synopsis

```
#include <curses.h>

int ins_wch(const cchar_t *wch);

int wins_wch(WINDOW *win, 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);
```

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

Return Value

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

Errors

No errors are defined.

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.

See Also

`add_wch()`, `<curses.h>`.

ins_wstr()

Name

`ins_wstr` - insert a wide-character string into the current window

Synopsis

```
#include <curses.h>

int ins_wstr(const wchar_t *wstr);
```

Description

Refer to `ins_nwstr()`.

intrflush()

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 *bf* is a boolean that specifies whether pressing an interrupt key (interrupt, suspend or quit) will flush the output buffer associated with the current screen. The default for the option is inherited from the display driver settings. The *win* argument is ignored.

Return Value

Upon successful completion, `intrflush()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

The same effect is achieved outside Curses using the NOFLSH local mode flag specified in the XBD specification (General Terminal Interface).

See Also

<curses.h>.

in_wch()

Name

in_wch, mvin_wch, mvwin_wch, win_wch - input a complex character and rendition from a window

Synopsis

```
#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.

See Also

<curses.h>.

in_wchnstr()

Name

in_wchnstr, in_wchstr, mvin_wchnstr, mvin_wchstr, mvwin_wchnstr, mvwin_wchstr, win_wchnstr, win_wchstr - input an array of complex characters and renditions from a window

Synopsis

```
#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()* 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.

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.

See Also

in_wch(), < curses.h >.

inwstr()

Name

inwstr - input a string of wide characters from the current window

Synopsis

```
#include <curses.h>

int inwstr(wchar_t *wstr);
```

Description

Refer to *innwstr()*.

isendwin()

Name

isendwin - determine whether a screen has been refreshed

Synopsis

```
#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.

See Also

`endwin()`, `<curses.h>`.

is_linetouched()

Name

`is_linetouched`, `is_wintouched`, `touchline`, `touchwin`, `untouchwin`, `wtouchln` - window refresh control functions

Synopsis

```
#include <curses.h>

bool is_linetouched(WINDOW *win, int line);
bool is_wintouched(WINDOW *win);
int touchline(WINDOW *win, int start, int count);
int touchwin(WINDOW *win);
int untouchwin(WINDOW *win);
int wtouchln(WINDOW *win, int y, int n, int changed);
```

Description

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

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.

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.

See Also

`doupdate()`, `<curses.h>`.

keyname()

Name

`keyname`, `key_name` - get name of key

Synopsis

```
#include <curses.h>

char *keyname(int c);

char *key_name(wchar_t c);
```

Description

The `keyname()` and `key_name()` 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 (<code>keyname()</code> only)	M-X
Key value defined in <code><curses.h></code> (<code>keyname()</code> 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, `keyname()` returns a pointer to a string as described above. Otherwise, it returns a null pointer.

Errors

No errors are defined.

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.

See Also

`meta()`, `<curses.h>`.

keypad()

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, `keypad()` returns `OK`. Otherwise, it returns `ERR`.

Errors

No errors are defined.

See Also

`<curses.h>`.

killchar()

Name

killchar, killwchar - terminal environment query functions

Synopsis

```
#include <curses.h>

char killchar(void);
```

```
int killwchar(wchar_t *ch);
```

Description

Refer to erasechar().

leaveok()

Name

leaveok - control cursor position resulting from refresh operations

Synopsis

```
#include <curses.h>

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

Description

Refer to clearok().

LINES

Name

LINES - number of lines on terminal screen

Synopsis

```
#include <curses.h>

extern int LINES;
```

Description

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

See Also

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

longname()

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, `longname()` returns a pointer to the description specified above. Otherwise, it returns a null pointer on error.

Errors

No errors are defined.

Application Usage

The return value of `longname()` may point to a static area which is overwritten by a subsequent call to `newterm()`.

See Also

`initscr()`, **<curses.h>**.

meta()

Name

meta - enable/disable meta-keys

Synopsis

```
#include <curses.h>

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

Description

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the display driver (see the XBD specification, General Terminal Interface). To force 8 bits to be returned, invoke `meta(win, TRUE)`. To force 7 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, `meta()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

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.

See Also

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

move()

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

The `move()` and `wmove()` 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.

See Also

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

mv

Name

`mv` - pointer page for functions with mv prefix

Description

Most cases in which a Curses function has the mv prefix ¹ 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.)

¹ The `mvcur()`, `mvderwin()` 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>addnstr()</i>
<i>mvaddnwstr()</i>	<i>mvwaddnwstr()</i>	<i>addnwstr()</i>
<i>mvaddwstr()</i>	<i>mvwaddwstr()</i>	<i>addnwstr()</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_wchnstr()</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>getnstr()</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>getn_wstr()</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>inchnstr()</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>insnstr()</i>
<i>mvinstr()</i>	<i>mvwinstr()</i>	<i>innstr()</i>
<i>mvins_nwstr()</i>	<i>mvwins_nwstr()</i>	<i>ins_nwstr()</i>
<i>mvins_wch()</i>	<i>mvwins_wch()</i>	<i>ins_wch()</i>
<i>mvins_wstr()</i>	<i>mvwins_wstr()</i>	<i>ins_nwstr()</i>
<i>mvinwstr()</i>	<i>mvwinwstr()</i>	<i>innwstr()</i>
<i>mvin_wch()</i>	<i>mvwin_wch()</i>	<i>in_wch()</i>
<i>mvin_wchnstr()</i>	<i>mvwin_wchnstr()</i>	<i>in_wchnstr()</i>

Function

mvin_wchstr()
mvprintw()
mvscanw()
mvvline()
mvvline_set()

mvwin_wchstr()
mvwprintw()
mvwscanw()
mvwvline()
mvwvline_set()

Refer to

in_wchnstr()
amvprintw()
mvscanw()
hline()
hline_set()

See Also

w.

mvcur()**Name**

mvcur - output cursor movement commands to the terminal

Synopsis

```
#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 move 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, *mvcur()* returns OK.

Otherwise, it returns ERR.

Errors

No errors are defined.

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.

See Also

doupdate(), *is_linetouched()*, **<curses.h>**.

mvderwin()

Name

mvderwin - define window coordinate transformation

Synopsis

```
#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, `mvderwin()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

See Also

derwin(), *doupdate()*, *dupwin()*, **<curses.h>**.

mvprintw()

Name

mvprintw, mvwprintw, printw, wprintw - print formatted output in window

Synopsis

```
#include <curses.h>

int mvprintw(int y, int x, char *fmt, ...);
int mvwprintw(WINDOW *win, int y, int x, char *fmt, ...);
int printw(char *fmt, ...);
int wprintw(WINDOW *win, char *fmt, ...);
```

Description

The `mvprintw()`, `mvwprintw()`, `printw()` and `wprintw()` 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.

See Also

addnstr(), *fprintf()*, **<curses.h>**

mvscanw()

Name

`mvscanw`, `mvwscanw`, `scanw`, `wscanw` - convert formatted input from a window

Synopsis

```
#include <curses.h>

int mvscanw(int y, int x, char *fmt, ...);
int mvwscanw(WINDOW *win, int y, int x, char *fmt, ...);
int scanw(char *fmt, ...);
int wscanw(WINDOW *win, 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.

See Also

getnstr(), *printw()*, *fscanf()*, *wcstombs()*, **<curses.h>**.

mvwin()

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, `mvwin()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

The application should not move subwindows by calling `mvwin()`.

See Also

`derwin()`, `doupdate()`, `is_linetouched()`, **<curses.h>**.

napms()

Name

`napms` - suspend the calling process

Synopsis

```
#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.

Application Usage

A more reliable method of achieving a timed delay is the `usleep()` function.

See Also

`delay_output()`, `usleep()` < curses.h >.

newpad()

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);

WINDOW *subpad(WINDOW *orig, int nlines, int ncols, int begin_y,
                int begin_x);
```

Description

The `newpad()` function creates a specialized WINDOW data structure representing a pad with *nlines* lines and *ncols* columns. A pad is like a window, except that it is not necessarily associated with a viewable part of the screen. Automatic refreshes of pads do not occur.

The `subpad()` function creates a subwindow within a pad with *nlines* lines and *ncols* columns. Unlike `subwin()`, which uses screen coordinates, the window is at position (*begin_y*, *begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window affect both windows.

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

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, `pnoutrefresh()` and `prefresh()` return OK. Otherwise, they return ERR.

Errors

No errors are defined.

Application Usage

To refresh a pad, call `prefresh()` or `pnoutrefresh()`, not `wrefresh()`. When porting code to use pads from WINDOWS, remember that these functions require additional arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display.

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 `refresh()`.

See Also

`derwin()`, `doupdate()`, `is_linetouched()`, **<curses.h>**.

newterm()

Name

`newterm` - screen initialization function

Synopsis

```
#include <curses.h>

SCREEN *newterm(char *type, FILE *outfile, FILE *infile);
```

Description

Refer to `initscr()`.

newwin()

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()`.

nl()

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.

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.

See Also

< curses.h >.

no

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.² They are found on 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>

nodelay()

Name

nodelay - enable or disable block during read

² 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.

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, `nodelay()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

See Also

getch(), *halfdelay()*, **<curses.h>**.

noqiflush()

Name

`noqiflush`, `qiflush` - enable/disable queue flushing

Synopsis

```
#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()` 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.

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).

See Also

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

notimeout()

Name

notimeout, timeout, wtimeout - control blocking on input

Synopsis

```
#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*:

<i>delay</i> < 0	One or more blocking reads (indefinite waits for input) are used.
<i>delay</i> = 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.
<i>delay</i> > 0	Any Curses input function blocks for <i>delay</i> 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.

See Also

getch(), *halfdelay()*, *nodelay()*, **<curses.h>**.

overlay()

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

The `overlay()` and `overwrite()` 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.

See Also

`copywin()`, `<curses.h>`.

pair_content()

Name

`pair_content`, `PAIR_NUMBER` - get information on a color pair

Synopsis

```
#include <curses.h>

int pair_content(short pair, short *f, short *b);

int PAIR_NUMBER(int value);
```

Description

Refer to `can_change_color()`.

pechochar()

Name

`pechochar`, `pecho_wchar` - write a character and rendition and immediately refresh the pad

Synopsis

```
#include <curses.h>

int pechochar(WINDOW *win, 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.

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.

See Also

`echochar()`, `echo_char()`, `newpad()`, **<curses.h>**.

pnoutrefresh()

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()`.

printw()

Name

`printw` - print formatted output in the current window

Synopsis

```
#include <curses.h>

int printw(char *fmt, ...);
```

Description

Refer to `mvprintw()`.

putp()

Name

`putp`, `tputs` - output commands to the terminal

Synopsis

```
#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 *fil*des specified in `setupterm()`.

The `tputs()` function outputs *str* to the terminal. The *str* argument must be a terminfo string variable or the return value from `tgetstr()`, `tgoto()`, `tigetstr()` 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.

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.

See Also

doupdate(), *is_linetouched()*, *putchar()*, *tgetent()*, *tigetflag()*, **<term.h>**.

putwin()

Name

putwin - dump window to a file

Synopsis

```
#include <curses.h>

int putwin(WINDOW *win, FILE *filep);
```

Description

Refer to *getwin()*.

qiflush()

Name

qiflush - enable queue flushing

Synopsis

```
#include <curses.h>

void qiflush(void);
```

Description

Refer to *noqiflush()*.

raw()

Name

raw - set Raw Mode

Synopsis

```
#include <curses.h>

int raw(void);
```

Description

Refer to *cbreak()*.

redrawwin()

Name

redrawwin, wredrawln - line update status functions

Synopsis

```
#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.

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.

See Also

`clearok()`, `doupdate()`, **<curses.h>**.

refresh()

Name

refresh - refresh current window

Synopsis

```
#include <curses.h>

int refresh(void);
```

Description

Refer to `doupdate()`.

reset_prog_mode()

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().

resetty()

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.

See Also

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

restartterm()

Name

restartterm - change terminal type

Synopsis

```
#include <term.h>

int restartterm(char *term, int fildes, int *errret);
```

Description

Refer to `del_curterm()`.

riponline()

Name

riponline - reserve a line for a dedicated purpose

Synopsis

```
#include <curses.h>

int riponline(int line, int (*init)(WINDOW *win, int columns));
```

Description

The `riponline()` function reserves a screen line for use by the application.

Any call to `riponline()` 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 `riponline()` above this limit have no effect but report success.

Return Value

The `riponline()` function returns OK.

Errors

No errors are defined.

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 `riponline()`. Thus, portable applications that use soft label functions should not call `riponline()` 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 `riponline()` ripped off. Portable applications should verify that *win* is not NULL before performing any operation on the window it represents.

See Also

`doupdate()`, `initscr()`, `slk_attroff()`, **<curses.h>**.

savetty()

Name

savetty - save terminal mode

Synopsis

```
#include <curses.h>

int savetty(void);
```

Description

Refer to resetty().

scanw()

Name

scanw - convert formatted input from the current window

Synopsis

```
#include <curses.h>

int scanw(char *fmt, ...);
```

Description

Refer to mvscanw().

scr_dump()

Name

scr_dump, scr_init, scr_restore, scr_set - screen file input/output functions

Synopsis

```
#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.

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()`.

See Also

`delscreen()`, `doupdate()`, `endwin()`, `getwin()`, `open()`, `read()`, `write()`, **<curses.h>**

scr1()

Name

`scr1`, `scroll`, `wscr1` - scroll a Curses window

Synopsis

```
#include <curses.h>

int scr1(int n);

int scroll(WINDOW *win);

int wscr1(WINDOW *win, int n);
```

Description

The `scroll()` function scrolls `win` one line in the direction of the first line.

The `scr1()` and `wscr1()` 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.

Return Value

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

Errors

No errors are defined.

See Also

<curses.h>.

scrollok()

Name

scrollok - enable or disable scrolling on a window

Synopsis

```
#include <curses.h>

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

Description

Refer to clearok().

setcchar()

Name

setcchar - set cchar_t from a wide character string and rendition

Synopsis

```
#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 edition of this document. Currently, the application must provide a null pointer as *opts*.

Return Value

Upon successful completion, setcchar() returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

See Also

attroff(), can_change_color(), getcchar(), <curses.h>.

set_curterm()

Name

set_curterm - set current terminal

Synopsis

```
#include <term.h>

TERMINAL *set_curterm(TERMINAL *nterm);
```

Description

Refer to del_curterm().

setscrreg()

Name

setscrreg, wsetscrreg - define software scrolling region

Synopsis

```
#include <curses.h>

int setscrreg(int top, int bot);

int wsetscrreg(WINDOW *win, int top, int bot);
```

Description

Refer to clearok().

set_term()

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, set_term() returns a pointer to the previous screen. Otherwise, it returns a null pointer.

Errors

No errors are defined.

Application Usage

This is the only function that manipulates SCREEN pointers; all other functions affect only the current screen.

See Also

initscr(), <curses.h>.

setupterm()

Name

setupterm - access the terminfo database

Synopsis

```
#include <term.h>

int setupterm(char *term, int fildes, int *errret);
```

Description

Refer to *del_curterm()*.

slk_attroff()

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

```
#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()` and `slk_attr_set()`, and `slk_color()` functions correspond to `slk_attroff()`, `slk_attron()`, `slk_attrset()` and `color_set()` and thus support the attribute constants with `WA_` prefix and `color`.

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

Return Value

Upon successful completion, `slk_label()` returns the requested label with leading and trailing blanks stripped. Otherwise, it returns a null pointer.

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

Errors

No errors are defined.

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.

See Also

*`attr_get()`, `attroff()`, `delscreen()`, `mbstowcs()`, `ripoffline()`, `wcswidth()`, **<curses.h>**.*

standend()

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.

See Also

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

start_color()

Name

start_color - initialize use of colors on terminal

Synopsis

```
#include <curses.h>

int start_color(void);
```

Description

Refer to can_change_color().

stdscr

Name

stdscr - default window

Synopsis

```
#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().

See Also

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

subpad()

Name

subpad - create a subwindow in a pad

Synopsis

```
#include <curses.h>

WINDOW *subpad(WINDOW *orig, int nlines, int ncols, int begin_y,
               int begin_x);
```

Description

Refer to newpad().

subwin()

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()`.

syncok()

Name

syncok, wcursyncup, wsyncdown, wsyncup - synchronise a window with its parents or children

Synopsis

```
#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, `syncok()` returns OK. Otherwise, it returns ERR.

The other functions do not return a value.

Errors

No errors are defined.

Application Usage

Applications seldom call `wsyncdown()` because it is called by all refresh operations.

See Also

doupdate(), *is_linetouched()*, **<curses.h>**.

termattrs()

Name

termattrs - get supported terminal video attributes

Synopsis

```
#include <curses.h>

chtype termattrs(void);

attr_t term_attr(void);
```

Description

The `termattrs()` function extracts the video attributes of the current terminal which is supported by the `chtype` data type.

The `term_attr()` 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_attr()` function returns a logical OR of `WA_` values of all video attributes supported by the terminal.

Errors

No errors are defined.

See Also

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

termname()

Name

termname - get terminal name

Synopsis

```
#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.

See Also

`del_curterm()`, `getenv()` `initscr()`, **<curses.h>**.

tgetent()

Name

`tgetent`, `tgetflag`, `tgetnum`, `tgetstr`, `tgoto` - termcap database emulation (TO BE WITHDRAWN)

Synopsis

```
#include <term.h>

int tgetent(char *bp, const char *name);
int tgetflag(char id[2]);
int tgetnum(char id[2]);
char *tgetstr(char id[2], char **area);
char *tgoto(char *cap, int col, int row);
```

Description

The `tgetent()` function looks up the termcap entry for *name*. The emulation ignores the buffer pointer *bp*.

The `tgetflag()` function gets the boolean entry for *id*.

The `tgetnum()` function gets the numeric entry for *id*.

The `tgetstr()` function gets the string entry for *id*. If *area* is not a null pointer and does not point to a null pointer, `tgetstr()` copies the string entry into the buffer pointed to by **area* and advances the variable pointed to by *area* to the first byte after the copy of the string entry.

The `tgoto()` function instantiates the parameters *col* and *row* into capability *cap* and returns a pointer to the resulting string.

All of the information available in the terminfo database need not be available through these functions.

Return Value

Upon successful completion, functions that return an integer return OK. Otherwise, they return ERR.

Functions that return pointers return a null pointer on error.

Errors

No errors are defined.

Application Usage

These functions are included as a conversion aid for programs that use the termcap library. Their arguments are the same and the functions are emulated using the terminfo database.

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.

Any terminal capabilities from the terminfo database that cannot be retrieved using these interfaces can be retrieved using the interfaces described on the `tigetflag()` page.

Portable applications must use `tputs()` to output the strings returned by `tgetstr()` and `tgoto()`.

See Also

`putc()`, `setupterm()`, `tigetflg()`, `<term.h>`.

tigetflag()

Name

`tigetflag`, `tigetnum`, `tigetstr`, `tparm` - retrieve capabilities from the terminfo database

Synopsis

```
#include <term.h>

int tigetflag(char *capname);

int tigetnum(char *capname);

char *tigetstr(char *capname);

char *tparm(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.

The `tparm()` function takes as *cap* a string capability. If *cap* is parameterized, `tparm()` resolves the parameterization. If the parameterized string refers to parameters *%p1* through *%p9*, then `tparm()` substitutes the values of *p1* through *p9*, respectively.

Return Value

Upon successful completion, `tigetflg()`, `tigetnum()` and `tigetstr()` 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.

Upon successful completion, `tparm()` returns *str* with parameterization resolved. Otherwise, it returns a null pointer.

Errors

No errors are defined.

Application Usage

For parameterized string capabilities, the application should pass the return value from `tigetstr()` to `tparm()`, as described above.

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:

Enhanced Curses

- 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 `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 `tparm()` cannot parse these user-defined strings.

See Also

def_prog_mode(), *tgetent()*, *putp()*, **<term.h>**.

timeout()

Name

timeout - control blocking on input

Synopsis

```
#include <curses.h>

void timeout(int delay);
```

Description

Refer to `notimeout()`.

touchline()

Name

touchline, touchwin - window refresh control functions

Synopsis

```
#include <curses.h>

int touchline(WINDOW *win, int start, int count);

int touchwin(WINDOW *win);
```

Description

Refer to `is_linetouched()`.

tparm()

Name

tparm - retrieve capabilities from the terminfo database

Synopsis

```
#include <term.h>

char *tparm(char *cap, long p1, long p2, long p3, long p4,
            long p5, long p6, long p7, long p8, long p9);
```

Description

Refer to tigetflag().

tputs()

Name

tputs - output commands to the terminal

Synopsis

```
#include <curses.h>

int tputs(const char *str, int affcnt, int (*putfunc)(int));
```

Description

Refer to putp().

typeahead()

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, typeahead() returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

See Also

douupdate(), *getch()*, *initscr()*, **<curses.h>**.

unctrl()

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, *unctrl()* returns the generated string. Otherwise, it returns a null pointer.

Errors

No errors are defined.

See Also

keyname(), *wunctrl()*, **<unctrl.h>**.

ungetch()

Name

ungetch, unget_wch - push a character onto the input queue

Synopsis

```
#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. If these functions are called too many times without an intervening call to *getch()* or *get_wch()*, the operation may fail.

Return Value

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

Errors

No errors are defined.

See Also

getch(), *get_wch()*, **<curses.h>**.

untouchwin()

Name

untouchwin - window refresh control function

Synopsis

```
#include <curses.h>

int untouchwin(WINDOW *win);
```

Description

Refer to *is_linetouched()*.

use_env()

Name

use_env - specify source of screen size information

Synopsis

```
#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.

See Also

del_curterm(), *initscr()*, **<curses.h>**.

vidattr()

Name

vidattr, vid_attr, vidputs, vid_puts - output attributes to the terminal

Synopsis

```
#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_pair_number, void *opt, int_t
              (*putfunc)(init_t));
```

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 that 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 edition of this document. Currently, the application must provide a null pointer as *opts*.

The user-supplied function *putfunc* (specified as an argument to `vidputs()`) is either `putchar()` or some other function with the same prototype. The `vidputs()` function ignores the return value of *putfunc*.

The `vid_attr()` and `vid_puts()` functions correspond to `vidattr()` and `vidputs()`, respectively, but take an argument of type `attr_t` and thus support the attribute constants with the `WA_` prefix.

The user-supplied function *putwfunc* (specified as an argument to `vid_puts()`) is either `putwchar()` or some other function with the same prototype. The `vid_puts()` function ignores the return value of *putwfunc*.

Return Value

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

Errors

No errors are defined.

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.

See Also

doupdate(), *is_linetouched()*, *putchar()*, *putwchar()*, *tigetflag()*, **<curses.h>**.

vline()

Name

vline - draw vertical line

Synopsis

```
#include <curses.h>

int vline(chtype ch, int n);
```

Description

Refer to *hline()*.

vline_set()

Name

vline_set - draw vertical line from complex character and rendition

Synopsis

```
#include <curses.h>

int vline_set(const cchar_t *ch, int n);
```

Description

Refer to *hline_set()*.

vwprintw()

Name

vwprintw - print formatted output in window

Synopsis

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

int vwprintw(WINDOW *, char *, va_list varglst);
```

Description

The *vwprintw()* function achieves the same effect as *wprintw()* using a variable argument list. The third argument is a *va_list*, as defined in **<varargs.h>**.

Return Value

Upon successful completion, `vwprintw()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

The `vwprintw()` function is deprecated because it relies on deprecated functions in the XSH specification. The `vw_printw()` function is preferred. The use of the `vwprintw()` and the `vw_printw()` functions in the same file will not work, due to the requirements to include **`varargs.h`** and **`stdarg.h`** which both contain definitions of `va_list`.

See Also

`mvprintw()`, `fprintf()`, `vw_printw()`, **`<curses.h>`**, **`<varargs.h>`**.

vw_printw()

Name

`vw_printw` - print formatted output in window

Synopsis

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

int vw_printw(WINDOW *, char *, va_list varglist);
```

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, `vw_printw()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

The `vw_printw()` function is preferred over `vwprintw()`. The use of the `vwprintw()` and the `vw_printw()` functions in the same file will not work, due to the requirement to include **`varargs.h`** and **`stdarg.h`** which both contain definitions of `va_list`.

See Also

`mvprintw()`, `fprintf()`, **`<curses.h>`**, **`<stdarg.h>`**.

vwscanw()

Name

vwscanw - convert formatted input from a window

Synopsis

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

int vwscanw(WINDOW *, char *, va_list varglist);
```

Description

The `vwscanw()` function achieves the same effect as `wscanw()` using a variable argument list. The third argument is a *va_list*, as defined in **<varargs.h>**.

Return Value

Upon successful completion, `vwscanw()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

The `vwscanw()` function is deprecated because it relies on deprecated functions in the XSH specification. The `vw_scanw()` function is preferred. The use of the `vwscanw()` and the `vw_scanw()` functions in the same file will not work, due to the requirement to include **varargs.h** and **stdarg.h** which both contain definitions of *va_list*.

See Also

fscanf(), *mvscanw()*, *vw_scanw()*, **<curses.h>**, **varargs.h**.

vw_scanw()

Name

vw_scanw - convert formatted input from a window

Synopsis

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

int vw_scanw(WINDOW *, 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, `vw_scanw()` returns OK. Otherwise, it returns ERR.

Errors

No errors are defined.

Application Usage

The `wscanw()` function is preferred over `vwscanw()`. The use of the `vwscanw()` and the `wscanw()` functions in the same file will not work, due to the requirement to include **`varargs.h`** and **`stdarg.h`** which both contain definitions of `va_list`.

See Also

`fscanf()`, `mvscanw()`, **`<curses.h>`**, **`<stdarg.h>`**.

W

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.³ (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 found on the following entries:

Function	Refer to
* There is no corresponding function without the w prefix.	
<code>waddch()</code>	<code>addch()</code>
<code>waddchnstr()</code>	<code>addchnstr()</code>
<code>waddchstr()</code>	<code>addchstr()</code>
<code>waddnstr()</code>	<code>addnstr()</code>
<code>waddstr()</code>	<code>addnstr()</code>
<code>waddnwstr()</code>	<code>addnwstr()</code>
<code>waddwstr()</code>	<code>addnwstr()</code>
<code>wadd_wch()</code>	<code>add_wch()</code>
<code>wadd_wchnstr()</code>	<code>add_wchnstr()</code>
<code>wadd_wchstr()</code>	<code>add_wchnstr()</code>
<code>wattroff()</code>	<code>attroff()</code>
<code>wattron()</code>	<code>attroff()</code>
<code>wattrset()</code>	<code>attroff()</code>
<code>wattr_get()</code>	<code>attr_get()</code>
<code>wattr_off()</code>	<code>attr_get()</code>
<code>wattr_on()</code>	<code>attr_get()</code>
<code>wattr_set()</code>	<code>attr_get()</code>

³ The `wunctrl()` function is an exception to this rule and has an entry under its own name.

Function	Refer to
<i>wbkgd()</i>	<i>bkgd()</i>
<i>wbkgdset()</i>	<i>bkgd()</i>
<i>wbkgrnd()</i>	<i>bkgrnd()</i>
<i>wbkgrndset()</i>	<i>bkgrnd()</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>wclrtobot()</i>	<i>clrtobot()</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>
<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>

Function	Refer to
<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>mvprintw()</i>
<i>wredrawln()</i>	<i>redrawln()</i>
<i>wrefresh()</i>	<i>doupdate()</i>
<i>wscanw()</i>	<i>mvscanw()</i>
<i>wscr1()</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_linetouch()</i>
<i>wvline()</i>	<i>hline()</i>
<i>wvline_set()</i>	<i>hline_set()</i>

wunctrl()

Name

wunctrl - generate printable representation of a wide character

Synopsis

```
#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, `wunctrl()` returns the generated string. Otherwise, it returns a null pointer.

Errors

No errors are defined.

See Also

keyname(), *unctrl()*, **<curses.h>**.

Chapter 6. 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 6, “Headers,” on page 121](#) 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.

<curses.h>

Name

curses.h - definitions for screen handling and optimization functions

Synopsis

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

Description

Objects

The <curses.h> header provides a declaration for COLOR_PAIRS, COLORS, COLS, curscr, LINES and stdscr.

Constants

The following constants are defined:

EOF

Function return value for end-of-file

ERR

Function return value for failure

FALSE

Boolean false value

OK

Function return value for success

TRUE

Boolean true value

WEOF

Wide-character function return value for end-of-file, as defined in <wchar.h>.

The following constant is defined if the implementation supports the indicated revision of the X/Open Curses specification.

```
_XOPEN_CURSES X/Open Curses, Issue 4 Version 2, May 1996, C610 <ISBN>  
(i.e. this document).
```

Data Types

The following data types are defined through **typedef**:

attr_t

An OR-ed set of attributes

CURSES

bool

Boolean data type

chtype

A character, attributes and a color-pair

SCREEN

An opaque terminal representation

wchar_t

As described in <stddef.h>

wint_t

As described in <wchar.h>

cchar_t

References a string of wide characters

WINDOW

An opaque window representation

The inclusion of <curses.h> may make visible all symbols from the headers <stdio.h>, <term.h>, <termios.h> and <wchar.h>.

Attribute Bits

The following symbolic constants 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 symbolic constants are used to manipulate attribute bits in objects of type **chtype**:

A_ALTCHARSET

Alternate character set

A_BLINK

Blinking

A_BOLD

Extra bright or bold

A_DIM

Half bright

A_INVIS

Invisible

A_PROTECT

Protected

A_REVERSE

Reverse video

A_STANDOUT

Best highlighting mode of the terminal

A_UNDERLINE

Underlining

These attribute flags need not be distinct except when `_XOPEN_CURSES` is defined and the application sets `_XOPEN_SOURCE_EXTENDED` to 1.

The following symbolic constants 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

A_COLOR

Bit-mask to extract color-pair information

The following symbolic constants 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

A_COLOR

Bit-mask to extract color-pair information

Line-Drawing Constants

The `<curses.h>` header defines the symbolic constants shown in the leftmost two columns of the following table for use in drawing lines. The symbolic constants that begin with `ACS_` are `char` constants. The symbolic constants that begin with `WACS_` are `cchar_t` constants for use 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.

char Constant	char_t Constant	POSIX Locale Default	Glyph Description
ACS_ULCORNER	WACS_ULCORNER	+	upper left-hand corner

CURSES

char Constant	char_t Constant	POSIX Locale Default	Glyph Description
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

char Constant	char_t Constant	POSIX Locale Default	Glyph Description
ACS_BLOCK	WACS_BLOCK	#	solid square block

Color-Related Macros

The following color-related macros are defined:

- COLOR_BLACK**
- COLOR_BLUE**
- COLOR_GREEN**
- COLOR_CYAN**
- COLOR_RED**
- COLOR_MAGENTA**
- COLOR_YELLOW**
- COLOR_WHITE**

Coordinate-Related Macros

The following coordinate-related macros are defined:

```
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 symbolic constants representing function key values are defined:

- | Key Code | Description |
|----------------------|--|
| KEY_CODE_YES | Used to indicate that a wchar_t variable contains a key code |
| KEY_BREAK | Break key |
| KEY_DOWN | Down arrow key |
| KEY_UP | Up arrow key |
| KEY_LEFT | Left arrow key |
| KEY_RIGHT | Right arrow key |
| KEY_HOME | Home key |
| KEY_BACKSPACE | Backspace |
| KEY_F0 | Function keys; space for 64 keys is reserved |
| KEY_F(n) | For 0_<n_<63 |

CURSES

KEY_DL

Delete line

KEY_IL

Insert line

KEY_DC

Delete character

KEY_IC

Insert char or enter insert mode

KEY_EIC

Exit insert char mode

KEY_CLEAR

Clear screen

KEY_EOS

Clear to end of screen

KEY_EOL

Clear to end of line

KEY_SF

Scroll 1 line forward

KEY_SR

Scroll 1 line backward (reverse)

KEY_NPAGE

Next page

KEY_PPAGE

Previous page

KEY_STAB

Set tab

KEY_CTAB

Clear tab

KEY_CATAB

Clear all tabs

KEY_ENTER

Enter or send

KEY_SRESET

Soft (partial) reset

KEY_RESET

Reset or hard reset

KEY_PRINT

Print or copy

KEY_LL

Home down or bottom

KEY_A1

Upper left of keypad

KEY_A3

Upper right of keypad

KEY_B2

Center of keypad

KEY_C1

Lower left of keypad

KEY_C3

Lower right of keypad

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 symbolic constant for a key code from the preceding table, such as KEY_A1.

The following symbolic constants representing function key values are also defined:

Key Code

Description

KEY_BTAB

Back tab key

KEY_BEG

Beginning key

KEY_CANCEL

Cancel key

KEY_CLOSE

Close key

KEY_COMMAND

Cmd (command) key

KEY_COPY

Copy key

KEY_CREATE

Create key

KEY_END

End key

KEY_EXIT

Exit key

KEY_FIND

Find key

KEY_HELP

Help key

KEY_MARK

Mark key

KEY_MESSAGE

Message key

KEY_MOVE

Move key

KEY_NEXT

Next object key

KEY_OPEN

Open key

KEY_OPTIONS

Options key

KEY_PREVIOUS

Previous object key

CURSES

KEY_REDO

Redo key

KEY_REFERENCE

Reference key

KEY_REFRESH

Refresh key

KEY_REPLACE

Replace key

KEY_RESTART

Restart key

KEY_RESUME

Resume key

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_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_SREDO

Shifted redo key

KEY_SREPLACE

Shifted replace key

KEY_SRIGHT

Shifted right arrow

KEY_SRSUME

Shifted resume key

KEY_SSAVE

Shifted save key

KEY_SSUSPEND

Shifted suspend key

KEY_SUNDO

Shifted undo key

KEY_SUSPEND

Suspend key

KEY_UNDO

Undo key

Function Prototypes

The following are declared as functions, and may also be defined as macros:

```

int     addch(const chtype);
int     addchstr(const chtype *, init);
int     addchnstr(chtype *const chstr, int n);
int     addchstr(const chtype *);
int     addnstr(const char *, init);
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);
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);
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 *, const cchar_t *);
int     box(WINDOW *, chtype, chtype);
int     box_set(WINDOW *, const cchar_t *, const cchar_t *);
bool    can_change_color(void);
int     cbreak(void);

```

```

int      chgat(int, attr_t, short, const void *);
int      clearok(WINDOW *, bool);
int      clear(void);
int      clrtoebot(WINDOW *win, bool bf);
int      clrtoeol(void);
int      color_content(short, short *, short *, short *);
int      COLOR_PAIR(int);
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);
void     delscreen(SCREEN *);
int      delwin(WINDOW *);
WINDOW   *derwin(WINDOW *, int, int, int, int);
int      doupdate(void);
WINDOW   *dupwin(WINDOW *);
int      echo(void);
int      echochar(const chtype);
int      echo_wchar(const cchar_t *);
int      endwin(void);
char     erasechar(void);
int      erase(void);
int      erasewchar(wchar_t *);
void     filter(void);
int      flash(void);
int      flushinp(void);
chtype   getbkgd(WINDOW *);
int      getbkgrnd(cchar_t *);
int      getcchar(const cchar_t *, wchar_t *, attr_t *,
                short *, void *);
int      getch(void);
int      getnstr(char *, int);
int      getn_wstr(wint_t *, int);
int      getstr(char *);
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);
int      hline(chtype, int);
int      hline_set(const cchar_t *, int);
void     idcok(WINDOW *, bool);
int      idlok(WINDOW *win, bool bf);
void     immedok(WINDOW *, bool);
chtype   inch(void);
int      inchnstr(chtype *, int);
int      inchstr(chtype *);
WINDOW   *initscr(void);
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);
int      insdelln(int);
int      insertln(void);
int      insnstr(const char *, int);
int      insstr(char *const str);
int      ins_nwstr(const wchar_t *, int);
int      insstr(const char *);
int      instr(char *);
int      ins_wch(const cchar_t *);
int      ins_wchstr(const cchar_t *);
int      intrflush(WINDOW *, bool);
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);
int      killwchar(wchar_t *);

```

```

int    leaveok(WINDOW *, bool);
char *longname(void);
int    meta(WINDOW *, bool);
int    move(int, int);
int    mvaddch(int, int, const chtype);
int    mvaddchnstr(int, int, const chtype *, int);
int    mvaddchstr(int, int, const chtype *);
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);
int    mvderwin(WINDOW *, int, int);
int    mvgetch(int, int);
int    mvgetnstr(int, int, char *, int);
int    mvgetn_wstr(int, int, wint_t *, int);
int    mvgetstr(int, int, char *);
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);
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);
int    mvinsnstr(int, int, const char *, int);
in     mvinsnwstr(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_watr(int, int, const wchar_t *);
in     mvin_wch(int, int, cchar_t *);
int    mvin_wchnstr(int, int, cchar_t *,);
int    mvin_wchstr(int, int, cchar_t *);
int    mvinwstr(int, int, wchar_t *);
int    mvprintw(int, int, char *, ...);
int    mvscanw(int, int, char *, ...);
int    mvvline(int, int, chtype, int);
int    mvvline_set(int, int, const cchar_t *, int);
int    mvwaddch(WINDOW *, int, int, const chtype);
int    mvwaddchnstr(WINDOW *, int, int, const chtype *, int);
int    mvwaddchstr(WINDOW *, int, int, const chtype *);
int    mvwaddnstr(WINDOW *, int, int, const char *, int);
int    mvwaddnwstr(WINDOW *, int, int, const wchar_t *, int);
int    mvwaddstr(WINDOW *, int, int, const char *);
int    mvwadd_wch(WINDOW *, int, int, const cchar_t *);
int    mvwadd_wchnstr(WINDOW *, int, int, const cchar_t *, int);
int    mvwadd_wchnstr(WINDOW *, int, int, const cchar_t *);
int    mvwaddwstr(WINDOW *, int, int, const wchar_t *);
int    mvwchgat(WINDOW *, int, int, int, attr_t,
               short, const void *);
int    mvwdelch(WINDOW *, int, int);
int    mvwgetch(WINDOW *, int, int);
int    mvwgetnstr(WINDOW *, int, int, char *, int);
int    mvwgetn_wstr(WINDOW *, int, int, wint_t *, int);
int    mvwgetstr(WINDOW *, int, int, char *);
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);
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);
int    mvwinsnstr(WINDOW *, int, int, const char *, int);
int    mvwinsnwstr(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, char *, ...);
int      mvwscanw(WINDOW *, int, int, char *, ...);
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(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);
void     noqiflush(void);
int      noraw(void);
int      notimeout(WINDOW *, bool);
int      overlay(const WINDOW *, WINDOW *);
int      overwrite(const WINDOW *, WINDOW *);
int      pair_content(short, short *, short *);
int      PAIR_NUMBER(int);
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      printfw(char *, ...);
int      putp(const char *);
int      putwin(WINDOW *, FILE *);
void     qiflush(void);
int      raw(void);
int      redrawwin(WINDOW *);
int      refresh(void);
int      resetty(void);
int      reset_prog_mode(void);
int      reset_shell_mode(void);
int      resetty(void);
int      ripoffline(int, int (*)(WINDOW *, int));
int      savetty(void);
int      scanw(char *, ...);
int      scr_dump(const char *);
int      scr_init(const char *);
int      scr_l(int);
int      scroll(WINDOW *);
int      scrollok(WINDOW *, bool);
int      scr_restore(const char *);
int      scr_set(const char *);
int      setcchar(cchar_t const wchar_t *, const attr_t,
                 short, const void *);
int      setscreg(int, int);
SCREEN  *set_term(SCREEN *);
int      setupterm(char *, int, int *);
int      slk_attr_off(const attr_t void *);
int      slk_attron(const attr_t void *);
int      slk_attron(const chtype);
int      slk_attr_set(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);
int      start_color(void);
WINDOW  *subpad(WINDOW *, int, int, int, int);
WINDOW  *subwin(WINDOW *, int, int, int, int);
int      syncok(WINDOW *, bool);
chtype  termattrs(void);
attr_t  term_attrs(void);
char     *termname(void);
int      tigetflag(char *);
int      tigetnum(char *);
char     *tigetstr(char *);
void     timeout(int);
int      touchline(WINDOW *, int, int);

```



```

int    touchwin(WINDOW *);
char *tparm(char *, long, long, long, long, long, long,
            long, long, long);
int    typeahead(int);
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 attr, short, void *, int (*)(int));
int    vidputs(chtype, int (*)(int));
int    vline(chtype, int);
int    vline_set(const cchar_t *, int);
int    vwprintw(WINDOW *, char *, va_list *);
int    vw_printw(WINDOW *, char *, va_list *);
int    wscanw(WINDOW *, char *, va_list *);
int    vw_scanw(WINDOW *, char *, va_list *);
int    waddch(WINDOW *, const chtype);
int    waddchnstr(WINDOW *, const chtype *, int);
int    waddchstr(WINDOW *, const chtype *);
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);
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    wbkggrnd(WINDOW *, const cchar_t *);
void   wbkggrndset(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 *, const cchar_t *);
int    wchgat(WINDOW *, int, attr_t, short, const void *);
int    wclear(WINDOW *);
int    wclrtoobot(WINDOW *);
int    wclrtoeol(WINDOW *);
void   wcursyncup(WINDOW *);
int    wcolor_set(WINDOW *, short, void *);
int    wdelch(WINDOW *);
int    wdeleteln(WINDOW *);
int    wechochar(WINDOW *, const chtype);
int    wecho_wchar(WINDOW *, const cchar_t *);
int    werase(WINDOW *);
int    wgetbkgrnd(WINDOW *, cchar_t *);
int    wgetch(WINDOW *);
int    wgetnstr(WINDOW *, char *, int);
int    wgetn_wstr(WINDOW *, wint_t *, int);
int    wgetstr(WINDOW *, char *);
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 *);
int    winchnstr(WINDOW *, chtype *, int);
int    winchstr(WINDOW *, chtype *);
int    winnstr(WINDOW *, char *, int);
int    winnwstr(WINDOW *, wchar_t *, int);
int    winsch(WINDOW *, chtype);
int    winsdelln(WINDOW *, int);
int    winstln(WINDOW *);
int    winsnstr(WINDOW *, const char *, int);
int    winsnwstr(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 *, char *, ...);
int    wredrawln(WINDOW *, int, int);
int    wrefresh(WINDOW *);
int    wscanw(WINDOW *, char *, ...);
int    wscrln(WINDOW *, int);
int    wsetscrreg(WINDOW *, int, int);
int    wstandend(WINDOW *);
int    wstandout(WINDOW *);
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);
```

See Also

<stdio.h>, <term.h>, <termios.h>, <unctrl.h>, <wchar.h>.

<termh>

Name

term.h - terminal capabilities

Synopsis

```
#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 contains the variable names listed in the Variable column.

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    tgetent(char *, const char);
int    tgetflag(char *);
int    tgetnum(char *);
char *tgetstr(char *, char **);
char *tgoto(char *, int, int);
int    tigetflag(char *);
int    tigetnum(char *);
char *tigetstr(char *);
char *tparm(char *, long, long, long, long, long, long,
            long, long, long, long);
int    tputs(const char *, int, int (*)(int));
```

See Also

printf(), *putp()*, *tigetflag()*, *tgetent()*, <curses.h>.

<unctrl.h>

Name

unctrl.h - definitions for unctrl()

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);
```

See Also

unctrl(), <curses.h>.

Chapter 7. Terminfo Source Format (ENHANCED CURSES)

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.

X/Open-compliant implementations provide a facility that accepts source files in the format specified in this chapter as a means of entering information into the **terminfo** database. The facility for installing this information into the database is implementation-specific. 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.

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.

Source File Syntax

Source files can use the ISO 8859-1 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 1) 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 file naming conventions established by implementation-specific **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-specific **terminfo** compilation utilities.

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.

Capability names adhere to an informal length limit of five characters. Whenever possible, capability names are chosen to be the same as or similar to those specified by the ANSI X3.64-1979 standard. Semantics are also intended to match those of the ANSI 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>**.

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	1023 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	1000 bytes
Length of a string representing a numeric value	99 digits
Magnitude of a numeric value	0 up to and including 32767

An implementation may support higher limits than those specified above.

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_LINE4 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
                    | 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_BOOLEAN5
                   | START_FEATURE_LINE_NUMERIC6
                   | START_FEATURE_LINE_STRING7
                   ;
```

```
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 graph⁸ characters other than '','/' and '|'.
3. A LONGNAME may contain any print⁹ 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 '#'.
 - b. The '=' character.
 - c. A string which may contain any print characters other than ','.
7. White space immediately following a ',' is ignored.
8. Comments consist of <bol>, optional whitespace, a required '#', and a terminating <eol>.
9. A header line must begin in column one.
10. A feature line must not begin in column one.
11. Blank lines are ignored.

Defined Capabilities

X/Open 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 *tgetent()*.

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 *tput* command.

⁴ An ALIAS that begins in column one. This is handled by the lexical analyzer.

⁵ A BOOLEAN feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

⁶ A NUMERIC feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

⁷ A STRING feature that begins after column one but is the first feature on the feature line. This is handled by the lexical analyzer.

⁸ Graph characters are those characters for which *isgraph()* returns non-zero.

⁹ Print characters are those characters for which *isprint()* returns non-zero.

Termcap

Codes provided for compatibility with older applications. These codes are **TO BE WITHDRAWN**. Because of this, not all **Capnames** have **Termcap** codes.

Booleans

Variable	Capname	Termcap	Description
auto_left_margin	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
back_color_erase	bce	ut	Screen erased with background color
can_change	ccc	cc	Terminal can re-define existing color
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
col_addr_glitch	xhpa	YA	Only positive motion for hpa/mhpa caps
cpi_changes_res	cpix	YF	Changing character pitch changes resolution
cr_cancels_micro_mode	crxm	YB	Using cr turns off micro mode
dest_tabs_magic_sms0	xt	xt	Destructive tabs, magic sms0 char (t1061)
eat_newline_glitch	xenl	xn	Newline ignored after 80 columns (Concept)
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (e.g., dialup, switch)
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	Cursor is hard to see
has_meta_key	km	km	Has a meta key (shift, sets parity bit)
has_print_wheel	daisy	YC	Printer needs operator to change character set
has_status_line	hs	hs	Has extra "status line"
hue_lightness_saturation	hls	hl	Terminal uses only HLS color notation (Tektronix)
insert_null_glitch	in	in	Insert mode distinguishes nulls
lpi_changes_res	lpix	YG	Changing line pitch changes resolution
memory_above	da	da	Display may be retained above the screen
memory_below	db	db	Display may be retained below the screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff required
no_esc_ctlc	xb	xb	Beehive (f1=escape, f2=ctrl C)
no_pad_char	npc	NP	Pad character doesn't exist

Variable	Capname	Termcap	Description
non_dest_scroll_region	ndscr	ND	Scrolling region is nondestructive
non_rev_rmcup	nrrmc	NR	smcup does not reverse rmcup
over_strike	os	os	Terminal overstrikes on hard-copy terminal
prtr_silent	mc5i	5i	Printer won't echo on screen
row_addr_glitch	xvpa	YD	Only positive motion for vpa/mvpa caps
semi_auto_right_margin	sam	YE	Printing in last column causes cr
status_line_esc_ok	eslok	es	Escape can be used on the status line
tilde_glitch	hz	hz	Hazeltine; can't print tilde (˜)
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking

Numbers

Variable	Capname	Termcap	Description
bit_image_entwining	bitwin	Yo	Number of passes for each bit-map row
bit_image_type	bitype	Yp	Type of bit image device
buffer_capacity	bufsz	Ya	Number of bytes buffered before printing
buttons	btns	BT	Number of buttons on the mouse
columns	cols	co	Number of columns in a line
dot_horz_spacing	spinh	Yc	Spacing of dots horizontally in dots per inch
dot_vert_spacing	spinv	Yb	Spacing of pins vertically in pins per inch
init_tabs	it	it	Tabs initially every # spaces
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of columns in each label
lines	lines	li	Number of lines on a screen or a page
lines_of_memory	lm	lm	Lines of memory if > lines ; 0 means varies
max_attributes	ma	ma	Maximum combined video attributes terminal can display
magic_cookie_glitch	xmc	sg	Number of blank characters left by sms or rmso
max_colors	colors	Co	Maximum number of colors on the screen
max_micro_address	maddr	Yd	Maximum value in micro_..._address
max_micro_jump	mjump	Ye	Maximum value in parm_..._micro
max_pairs	pairs	pa	Maximum number of color-pairs on the screen
maximum_windows	wnum	MW	Maximum number of definable windows
micro_col_size	mcs	Yf	Character step size when in micro mode
micro_line_size	mls	Yg	Line step size when in micro mode

Variable	Capname	Termcap	Description
no_color_video	ncv	NC	Video attributes that can't be used with colors
num_labels	nlab	Nl	Number of labels on screen (start at 1)
number_of_pins	npins	Yh	Number of pins in print-head
output_res_char	orc	Yi	Horizontal resolution in units per character
output_res_line	orl	Yj	Vertical resolution in units per line
output_res_horz_inch	orhi	Yk	Horizontal resolution in units per inch
output_res_vert_inch	orvi	Yl	Vertical resolution in units per inch
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
print_rate	cps	Ym	Print rate in characters per second
virtual_terminal	vt	vt	Virtual terminal number
wide_char_size	widcs	Yn	Character step size when in double-wide mode
width_status_line	wsl	ws	Number of columns in status line

Strings

Variable	Capname	Termcap	Description
acs_chars	acsc	ac	Graphic charset pairs aAbBcC
alt_scancode_esc	scesa	S8	Alternate escape for scancode emulation (default is for VT100)
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
bit_image_carriage_return	bicr	Yv	Move to beginning of same row
bit_image_newline	binel	Zz	Move to next row of the bit image
bit_image_repeat	birep	Xy	Repeat bit-image cell #1 #2 times
carriage_return	cr	cr	Carriage return
change_char_pitch	cpi	ZA	Change number of characters per inch
change_line_pitch	lpi	ZB	Change number of lines per inch
change_res_horz	chr	ZC	Change horizontal resolution
change_res_vert	cvr	ZD	Change vertical resolution
change_scroll_region	csr	cs	Change to lines #1 through #2 (VT100)
char_padding	rmp	rP	Like ip but when in replace mode
char_set_names	csnm	Zy	Returns a list of character set names
clear_all_tabs	tbc	ct	Clear all tab stops
clear_margins	mgc	MC	Clear all margins (top, bottom, and sides)
clear_screen	clear	cl	Clear screen and home cursor
clr_bol	el1	cb	Clear to beginning of line, inclusive

Variable	Capname	Termcap	Description
<code>clr_eol</code>	el	ce	Clear to end of line
<code>clr_eos</code>	ed	cd	Clear to end of display
<code>code_set_init</code>	csin	ci	Init sequence for multiple codesets
<code>color_names</code>	colorm	Yw	Give name for color #1
<code>column_address</code>	hpa	ch	Set horizontal position to absolute #1
<code>command_character</code>	cmdch	CC	Terminal settable cmd character in prototype
<code>create_window</code>	cwin	CW	Define win #1 to go from #2,#3 to #4,#5
<code>cursor_address</code>	cup	cm	Move to row #1 col #2
<code>cursor_down</code>	cud1	do	Down one line
<code>cursor_home</code>	home	ho	Home cursor (if no cup)
<code>cursor_invisible</code>	civis	vi	Make cursor invisible
<code>cursor_left</code>	cub1	le	Move left one space.
<code>cursor_mem_address</code>	mrcup	CM	Memory relative cursor addressing
<code>cursor_normal</code>	cnorm	ve	Make cursor appear normal (undo vs/vi)
<code>cursor_right</code>	cuf1	nd	Non-destructive space (cursor or carriage right)
<code>cursor_to_ll</code>	ll	ll	Last line, first column (if no cup)
<code>cursor_up</code>	cuu1	up	Upline (cursor up)
<code>cursor_visible</code>	cvvis	vs	Make cursor very visible
<code>define_bit_image_region</code>	defbi	Yx	Define rectangular bit-image region
<code>define_char</code>	defc	ZE	Define a character in a character set
<code>delete_character</code>	dch1	dc	Delete character
<code>delete_line</code>	dl1	d1	Delete line
<code>device_type</code>	devt	dv	Indicate language/codeset support
<code>dial_phone</code>	dial	DI	Dial phone number #1
<code>dis_status_line</code>	dsl	ds	Disable status line
<code>display_clock</code>	dclk	DK	Display time-of-day clock
<code>display_pc_char</code>	dispc	S1	Display PC character
<code>down_half_line</code>	hd	hd	Half-line down (forward 1/2 linefeed)
<code>ena_acs</code>	enacs	eA	Enable alternate character set
<code>end_bit_image_region</code>	endbi	Yy	End a bit-image region
<code>enter_alt_charset_mode</code>	smacs	as	Start alternate character set
<code>enter_am_mode</code>	smam	SA	Turn on automatic margins
<code>enter_blink_mode</code>	blink	mb	Turn on blinking
<code>enter_bold_mode</code>	bold	md	Turn on bold (extra bright) mode
<code>enter_ca_mode</code>	smcup	ti	String to begin programs that use cup

Variable	Capname	Termcap	Description
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_doublewide_mode	swidm	ZF	Enable double wide printing
enter_draft_quality	sdrfq	ZG	Set draft quality print
enter_horizontal_hl_mode	ehhlm		Turn on horizontal highlight mode
enter_insert_mode	smir	im	Insert mode (enter)
enter_italics_mode	sitm	ZH	Enable italics
enter_left_hl_mode	elhlm		Turn on left highlight mode
enter_leftward_mode	slm	ZI	Enable leftward carriage motion
enter_low_hl_mode	elohlm		Turn on low highlight mode
enter_micro_mode	smicm	ZJ	Enable micro motion capabilities
enter_near_letter_quality	snlq	ZK	Set near-letter quality print
enter_normal_quality	snrmq	ZL	Set normal quality print
enter_pc_charset_mode	smpch	S2	Enter PC character display mode
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_right_hl_mode	erhlm		Turn on right highlight mode
enter_scancode_mode	smsc	S4	Enter PC scancode mode
enter_secure_mode	invis	mk	Turn on blank mode (characters invisible)
enter_shadow_mode	sshm	ZM	Enable shadow printing
enter_standout_mode	smsso	so	Begin standout mode
enter_subscript_mode	ssubm	ZN	Enable subscript printing
enter_superscript_mode	ssupm	ZO	Enable superscript printing
enter_top_hl_mode	ethlm		Turn on top highlight mode
enter_underline_mode	smul	us	Start underscore mode
enter_upward_mode	sum	ZP	Enable upward carriage motion
enter_vertical_hl_mode	evhlm		Turn on vertical highlight mode
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use cup
exit_delete_mode	rmdc	ed	End delete mode
exit_doublewide_mode	rwidm	ZQ	Disable double wide printing
exit_insert_mode	rmir	ei	End insert mode

Variable	Capname	Termcap	Description
exit_italics_mode	ritm	ZR	Disable italics
exit_leftward_mode	rlm	ZS	Enable rightward (normal) carriage motion
exit_micro_mode	rmicm	ZT	Disable micro motion capabilities
exit_pc_charset_mode	rmpch	S3	Disable PC character display mode
exit_scancode_mode	rmsc	S5	Disable PC scancode mode
exit_shadow_mode	rshm	ZU	Disable shadow printing
exit_standout_mode	rmsso	se	End standout mode
exit_subscript_mode	rsubm	ZV	Disable subscript printing
exit_superscript_mode	rsupm	ZW	Disable superscript printing
exit_underline_mode	rmul	ue	End underscore mode
exit_upward_mode	rum	ZX	Enable downward (normal) carriage motion
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
fixed_pause	pause	PA	Pause for 2-3 seconds
flash_hook	hook	fh	Flash the switch hook
flash_screen	flash	vb	Visible bell (may move cursor)
form_feed	ff	ff	Hardcopy terminal page eject
from_status_line	fsl	fs	Return from status line
get_mouse	getm	Gm	Curses should get button events
goto_window	wingo	WG	Go to window #1
hangup	hup	HU	Hang-up phone
init_1string	is1	i1	Terminal or printer initialization string
init_2string	is2	is	Terminal or printer initialization string
init_3string	is3	i3	Terminal or printer initialization string
init_file	if	if	Name of initialization file
init_prog	iprog	iP	Path name of program for initialization
initialize_color	initc	IC	Set color #1 to RGB #2, #3, #4
initialize_pair	initp	Ip	Set color-pair #1 to fg #2, bg #3
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line
insert_padding	ip	ip	Insert pad after character inserted
Note: The " key_ " strings are sent by specific keys. The " key_ " descriptions include the macro, defined in <curses.h> , for the code returned by <i>getch()</i> when the key is pressed (see <i>getch()</i>).			
key_a1	ka1	K1	upper left of keypad
key_a3	ka3	K3	upper right of keypad
key_b2	kb2	K2	center of keypad
key_backspace	kbs	kb	sent by backspace key

Variable	Capname	Termcap	Description
key_beg	kbeg	@1	sent by beg(inning) key
key_btab	kcbt	kB	sent by back-tab key
key_c1	kc1	K4	lower left of keypad
key_c3	kc3	K5	lower right of keypad
key_cancel	kcan	@2	sent by cancel key
key_catab	ktbc	ka	sent by clear-all-tabs key
key_clear	kclr	kC	sent by clear-screen or erase key
key_close	kclo	@3	sent by close key
key_command	kcmd	@4	sent by cmd (command) key
key_copy	kcpy	@5	sent by copy key
key_create	kcrt	@6	sent by create key
key_ctab	kctab	kt	sent by clear-tab key
key_dc	kdch1	kD	sent by delete-character key
key_d1	kd11	kL	sent by delete-line key
key_down	kcud1	kd	sent by terminal down-arrow key
key_eic	krmir	kM	sent by rmir or smir in insert mode
key_end	kend	@7	sent by end key
key_enter	kent	@8	sent by enter/send key
key_eol	kel	kE	sent by clear-to-end-of-line key
key_eos	ked	kS	sent by clear-to-end-of-screen key
key_exit	kext	@9	sent by exit key
key_f0	kf0	k0	sent by function key f0
key_f1	kf1	k1	sent by function key f1
:	:	..	:
key_f62	kf62	Fq	sent by function key f62
key_f63	kf63	Fr	sent by function key f63
key_find	kfnd	@0	sent by find key
key_help	khlp	%1	sent by help key
key_home	khome	kh	sent by home key
key_ic	kich1	kI	sent by ins-char/enter ins-mode key
key_il	kil1	kA	sent by insert-line key
key_left	kcub1	k1	sent by terminal left-arrow key
key_ll	kll	kH	sent by home-down key
key_mark	kmrk	%2	sent by mark key
key_message	kmsg	%3	sent by message key
key_mouse	kmous	Km	0631, Mouse event has occurred
key_move	kmov	%4	sent by move key

Variable	Capname	Termcap	Description
key_next	knxt	%5	sent by next-object key
key_npage	knp	kN	sent by next-page key
key_open	kopn	%6	sent by open key
key_options	kopt	%7	sent by options key
key_ppage	kpp	kP	sent by previous-page key
key_previous	kprv	%8	sent by previous-object key
key_print	kpri	%9	sent by print or copy key
key_redo	krdo	%0	sent by redo key
key_reference	kref	&1	sent by ref(erence) key
key_refresh	krfr	&2	sent by refresh key
key_replace	krpl	&3	sent by replace key
key_restart	krst	&4	sent by restart key
key_resume	kres	&5	sent by resume key
key_right	kcuf1	k1	sent by terminal right-arrow key
key_save	ksav	&6	sent by save key
key_sbeg	kBEG	&9	sent by shifted beginning key
key_scancel	kCAN	&0	sent by shifted cancel key
key_scommand	kCMD	*1	sent by shifted command key
key_scopy	kCPY	*2	sent by shifted copy key
key_screate	kCRT	*3	sent by shifted create key
key_sdc	kDC	*4	sent by shifted delete-char key
key_sdl	kDL	*5	sent by shifted delete-line key
key_select	kslt	*6	sent by select key
key_send	kEND	*7	sent by shifted end key
key_seol	KEOL	*8	sent by shifted clear-line key
key_sexit	kEXT	*9	sent by shifted exit key
key_sf	kind	kF	sent by scroll-forward/down key
key_sfind	kFND	*0	sent by shifted find key
key_shelp	kHLP	#1	sent by shifted help key
key_shome	kHOM	#2	sent by shifted home key
key_sic	kIC	#3	sent by shifted input key
key_sleft	kLFT	#4	sent by shifted left-arrow key
key_smessage	kMSG	%a	sent by shifted message key
key_smove	kMOV	%b	sent by shifted move key
key_snext	kNXT	%c	sent by shifted next key
key_soptions	kOPT	%d	sent by shifted options key
key_sprevious	kPRV	%e	sent by shifted prev key

Variable	Capname	Termcap	Description
key_sprint	kPRT	%f	sent by shifted print key
key_sr	kri	kR	sent by scroll-backward/up key
key_sredo	kRDO	%g	sent by shifted redo key
key_sreplace	kRPL	%h	sent by shifted replace key
key_sright	kRIT	%i	sent by shifted right-arrow key
key_sresume	kRES	%j	sent by shifted resume key
key_ssav	kSAV	!1	sent by shifted save key
key_ssuspend	kSPD	!2	sent by shifted suspend key
key_stab	khts	kT	sent by set-tab key
key_sundo	kUND	!3	sent by shifted undo key
key_suspend	kspd	&7	sent by suspend key
key_undo	kund	&8	sent by undo key
key_up	kcuu1	ku	sent by terminal up-arrow key
keypad_local	rmkx	ke	Out of "keypad-transmit" mode
keypad_xmit	smkx	ks	Put terminal in "keypad-transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_format	fln	Lf	Label format
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
micro_column_address	mhp	ZY	Like column_address for micro adjustment
micro_down	mcud1	ZZ	Like cursor_down for micro adjustment
micro_left	mcub1	Za	Like cursor_left for micro adjustment
micro_right	mcuf1	Zb	Like cursor_right for micro adjustment
micro_row_address	mvpa	Zc	Like row_address for micro adjustment

Variable	Capname	Termcap	Description
micro_up	mcuu1	Zd	Like cursor_up for micro adjustment
mouse_info	minfo	Mi	Mouse status information
newline	nel	nw	Newline (behaves like cr followed by lf)
order_of_pins	porder	Ze	Matches software bits to print-head pins
orig_colors	oc	oc	Set all color(-pair)s to the original ones
orig_pair	op	op	Set default color-pair to the original one
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars
parm_delete_line	dl	DL	Delete #1 lines
parm_down_cursor	cud	DO	Move down #1 lines.
parm_down_micro	mcud	Zf	Like parm_down_cursor for micro adjust.
parm_ich	ich	IC	Insert #1 blank chars
parm_index	indn	SF	Scroll forward #1 lines.
parm_insert_line	il	AL	Add #1 new blank lines
parm_left_cursor	cub	LE	Move cursor left #1 spaces
parm_left_micro	mcub	Zg	Like parm_left_cursor for micro adjust.
parm_right_cursor	cuf	RI	Move right #1 spaces.
parm_right_micro	mcuf	Zh	Like parm_right_cursor for micro adjust.
parm_rindex	rin	SR	Scroll backward #1 lines.
parm_up_cursor	cuu	UP	Move cursor up #1 lines.
parm_up_micro	mcuu	Zi	Like parm_up_cursor for micro adjust.
pc_term_options	pctrm	S6	PC terminal options
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	p1	Prog funct key #1 to execute string #2
pkey_plab	pfxl	x1	Prog key #1 to xmit string #2 and show string #3
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	p0	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
pulse	pulse	PU	Select pulse dialing
quick_dial	qdial	QD	Dial phone number #1, without progress detection

Variable	Capname	Termcap	Description
remove_clock	rmclk	RC	Remove time-of-day clock
repeat_char	rep	rp	Repeat char #1 #2 times
req_for_input	rfi	RF	Send next input char (for ptys)
req_mouse_pos	reqmp	RQ	Request mouse position report
reset_1string	rs1	r1	Reset terminal completely to sane modes
reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last sc
row_address	vpa	cv	Set vertical position to absolute #1
save_cursor	sc	sc	Save cursor position
scancode_escape	scesc	S7	Escape for scancode emulation
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
select_char_set	scs	Zj	Select character set
set0_des_seq	s0ds	s0	Shift into codeset 0 (EUC set 0, ASCII)
set1_des_seq	s1ds	s1	Shift into codeset 1
set2_des_seq	s2ds	s2	Shift into codeset 2
set3_des_seq	s3ds	s3	Shift into codeset 3
set_a_attributes	sgr1		Define second set of video attributes #1-#6
set_a_background	setab	AB	Set background color to #1 using ANSI escape
set_a_foreground	setaf	AF	Set foreground color to #1 using ANSI escape
set_attributes	sgr	sa	Define first set of video attributes #1-#9
set_background	setb	Sb	Set background color to #1
set_bottom_margin	smgb	Zk	Set bottom margin at current line
set_bottom_margin_parm	smgbp	Zl	Set bottom margin at line #1 or #2 lines from bottom
set_clock	sclk	SC	Set clock to hours (#1), minutes (#2), seconds (#3)
set_color_band	setcolor	Yz	Change to ribbon color #1
set_color_pair	scp	sp	Set current color pair to #1
set_foreground	setf	Sf	Set foreground color to #1
set_left_margin	smgl	ML	Set left margin at current column

Variable	Capname	Termcap	Description
set_left_margin_parm	smglp	Zm	Set left (right) margin at column #1 (#2)
set_lr_margin	smglr	ML	Sets both left and right margins
set_page_length	slines	YZ	Set page length to #1 lines
set_pglen_inch	slength	YI	Set page length to #1 hundredth of an inch
set_right_margin	smgr	MR	Set right margin at current column
set_right_margin_parm	smgrp	Zn	Set right margin at column #1
set_tab	hts	st	Set a tab in all rows, current column
set_tb_margin	smgtb	MT	Sets both top and bottom margins
set_top_margin	smgt	Zo	Set top margin at current line
set_top_margin_parm	smgtp	Zp	Set top (bottom) margin at line #1 (#2)
set_window	wind	wi	Current window is lines #1-#2 cols #3-#4
start_bit_image	sbim	Zq	Start printing bit image graphics
start_char_set_def	scsd	Zr	Start definition of a character set
stop_bit_image	rbim	Zs	End printing bit image graphics
stop_char_set_def	rcsd	Zt	End definition of a character set
subscript_characters	subcs	Zu	List of "subscript-able" characters
superscript_characters	supcs	Zv	List of "superscript-able" characters
tab	ht	ta	Tab to next 8-space hardware tab stop
these_cause_cr	docr	Zw	Printing any of these chars causes cr
to_status_line	tsl	ts	Go to status line, col #1
tone	tone	T0	Select touch tone dialing
user0	u0	u0	User string 0
user1	u1	u1	User string 1
user2	u2	u2	User string 2
user3	u3	u3	User string 3
user4	u4	u4	User string 4
user5	u5	u5	User string 5
user6	u6	u6	User string 6
user7	u7	u7	User string 7
user8	u8	u8	User string 8
user9	u9	u9	User string 9
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
wait_tone	wait	WA	Wait for dial tone
xoff_character	xoffc	XF	X-off character

Variable	Capname	Termcap	Description
xon_character	xonc	XN	X-on character
zero_motion	zerom	Zx	No motion for the subsequent character

Sample Entry

The following entry describes the AT&T; 610 terminal.

```
610|610bct|ATT610|att610|AT&T610;80column;98key; keyboard,
am, eslok, hs, mir, msg, xenl, xon,
cols#80, it#8, lh#2, lines#24, lw#8, nlab#8, wsl#80,
acsc=`aaffggjjkkllmmnnoppqrrssttuuvvwxxyzz{|}|~&tilde;&tilde;,
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[%p1%d;%p2%dr, cub=\E[%p1%dD, cub1=\b,
cud=\E[%p1%dB, cud1=\E[B, cuf=\E[%p1%dC, cuf1=\E[C,
cup=\E[%i%p1%d;%p2%dH, cuu=\E[%p1%dA, cuu1=\E[A,
cvvis=\E[12;25h, dch=\E[%p1%dP, dch1=\E[P, dim=\E[2m,
dl=\E[%p1%dM, dl1=\E[M, ed=\E[J, el=\E[K, el1=\E[1K,
flash=\E[5h<200>\E[5l, fsl=\E[8, home=\E[H, ht=\t,
ich=\E[%p1%d@, il=\E[%p1%dL, il1=\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, is3=\E(B\E)0, kLFT=\E[\s@, kRIT=\E[\sA,
kbs=.H, kcbt=\E[Z, kclr=\E[2J, kcu1=\E[D, kcu1=\E[B,
kcu1=\E[C, kcu1=\E[A, kfP=\E0c, kfP0=\ENp,
kfP1=\ENq, kfP2=\ENr, kfP3=\ENs, kfP4=\ENT, kfI=\E0d,
kfB=\E0e, kf4=\E0f, kf(CW=\E0g, kf6=\E0h, kf7=\E0i,
kf8=\E0j, kf9=\E0n, khome=\E[H, kind=\E[S, kri=\E[T,
ll=\E[24H, mc4=\E[4i, mc5=\E[5i, nel=\EE,
pfxl=\E[%p1%d;%p2%l%02dq%p1%{9}%< %t\s\s\sF%p1%1d\s\s\s\s
\s\s\s\s\s;%p2%2s,
pln=\E[%p1%d;0;0;0q%p2%:-16.16s, rc=\E8, rev=\E[7m,
ri=\EM, rmacs=0, 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,0%;,
sgr0=\E[m,0, smacs=N, smir=\E[4h, smln=\E[p,
smso=\E[7m, smul=\E[4m, ts1=\E7\E[25;%i%p1%dx,
```

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.

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-x (for any appropriate x)
\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** Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

Device Capabilities

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, cub1=_b, cnd1=_n,
dch1=_E[P, dl1=_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, cub1=_H,
cud1=_J, ind=_J, lines#24,
```

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 **mr cup**.

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, one would use %p1%{5}%-.

The % encodings have the following meanings:

%%

Outputs '%’.

%[:]flags[width[.precision]][doxXs]

As in printf(); flags are [-+#] and space.

%c

Print pop() gives %c.

%p[1-9]

Push the ith argument.

%P[a-z]

Set dynamic variable [a-z] to pop().

%g[a-z]

Get dynamic variable [a-z] and push it.

%P[A-Z]

Set static variable [a-z] to pop().

%g[A-Z]

Get static variable [a-z] and push it.

%'c'

Push char constant c.

%{nn}

Push decimal constant nn.

%l

Push strlen(pop()).

%+ %- %* %/ %m

Arithmetic (%m is mod): push(pop integer2 op pop integer1) where integer1 represents the top of the stack

%& %| %^

Bit operations: push(pop integer2 op pop integer1)

%= %> %<

Logical operations: push(pop integer2 op pop integer1)

%A %O

Logical operations: and, or

%! %˜

Unary operations: push(op pop())

%i

(For ANSI terminals) add 1 to the first argument (if one argument present), or first two arguments (if more than one argument present).

% expr %t thenpart %e elsepart %;

If-then-else, %e elsepart is optional; else-if's are possible ala Algol 68: % c1 %t b1 %e c2 %t b2 %e c3 %t b3 %e c4 %t b4 %e b5%; ci are conditions, bi 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 tty modes so that tabs 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.

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** sequence will not restore the screen after an **smcup** sequence is output (to the state prior to outputting **smcup**), specify **nrrmc**.

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.)

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 the **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.

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 blanks 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 blanks. 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 blanks 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 multiline 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* blanks.

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* blanks without moving the cursor) can be given as **ech** with one argument.

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 Teleray 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 new line 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 arguments. 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.

tparm 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 *bold* 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% %t;3%;

Sequence	When to Output	terminfo Translation
;4	if p1 or p3 or p6	%%p1%p3% p6% t;4%;
;5	if p4	%%p4%t;5%;
;7	if p1 or p5	%%p1%p5% t;7%;
;8	if p7	%%p7%t;8%;
m	always	m
caret.N or ^O	if p9 ^N, else ^O	%%p9%t^N%e^O%;

Putting this all together into the **sgr** sequence gives:

```
sgr=\E[0%%p2%p6%|t;3%;%%p1%p3%|p6%
|t;4%;%%p5%t;5%;%%p1%p5%
|t;7%;%%p7%t;8%;m%%p9%t,N%e,O%;,
```

Remember that **sgr** and **sgr0** must always be specified.

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**, **kcuf1**, **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 tabs), **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), **knp** (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 **pfx**. 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 **pfx** 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.

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 tabs 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 tabs 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 tabs 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 nonvolatile 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; **iprog**, 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 **iprog**; output **is1**; output **is2**; set the margins using **mgc**, **smgl** and **smgr**; set the tabs 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 tabs than can be described by using **tbc** 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.

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**.

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**.

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

Glyph Name	VT100+ Character
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.

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).

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, **tparm(repeat_char, 'x', 10)** is the same as **xxxxxxxxxx**.

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 **rfi**.

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 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th 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.

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 linefeed 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 tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). 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.

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.

Printer Capabilities

The **terminfo** database lets you define capabilities of printers as well as terminals.

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.

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.

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

Normal Mode:

orc	Steps moved horizontally
orl	Steps moved vertically

Micro Mode:

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	
cpi cpix	Change character pitch If set, cpi changes orhi, otherwise changes orc
lpi lpix	Change line pitch If set, lpi changes orvi, otherwise changes orl
chr cvr	Change steps per column Change steps per line

The **cpi** and **lpi** 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 **cpi**, **lpi**, **chr**, or **cvr** should recalculate the printer resolution (and should recalculate other values).

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 smallest steps.

String Capabilities for Motion	Description
mcub1 mcuf1 mcuu1 mcud1	Move 1 step left Move 1 step right Move 1 step up Move 1 step down
mcub mcuf mcuu mcud	Move <i>N</i> steps left Move <i>N</i> steps right Move <i>N</i> steps up Move <i>N</i> steps down
mhpa mvpa	Move <i>N</i> steps from the left Move <i>N</i> 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	Description
mjump maddr	Limit on use of mcub1 , mcuf1 , mcuu1 , mcud1 Limit on use of mhpa , mvpa
xhpa xvpa	If set, hpa and mhpa can't move left 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	Description
smicm rmicm	Enter micro mode 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	Description
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	Description
slm rlm sum rum	Reverse sense of horizontal motions Restore sense of horizontal motions Reverse sense of vertical motions Restore sense of vertical motions
<i>While sense of horizontal motions reversed:</i>	
mcub1 mcuf1 mcub mcuf cub1 cuf1 cub cuf	Move 1 step right Move 1 step left Move N steps right Move N steps left Move 1 column right Move 1 column left Move N columns right Move N columns left
<i>While sense of vertical motions reversed:</i>	
mcuu1 mcud1 mcuu mcud cuu1 cud1 cuu cud	Move 1 step down Move 1 step up Move N steps down Move N steps up Move 1 line down Move 1 line up Move N lines down Move N 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	Description
docr zerom	List of control characters causing cr 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	Description
smgl smgr smgb smgt	Set left margin at current column Set right margin at current column Set bottom margin at current line Set top margin at current line
smgbp smglp smgrp smgtp	Set bottom margin at line N Set left margin at column N Set right margin at column N Set top margin at line N

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 *M*, 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 **smglp** 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	Description
sshm rshm	Enter shadow-printing mode Exit shadow-printing mode
sitm ritm	Enter italicizing mode Exit italicizing mode
swidm rwidm	Enter wide character mode Exit wide character mode

Enhanced Printing	Description
ssupm rsupm supcs	Enter superscript mode Exit superscript mode List of characters available as superscripts
ssubm rsubm subcs	Enter subscript mode Exit subscript mode List of characters available as subscripts

If a printer requires the **sshm** control sequence before every character to be shadow-printed, the **rshm** string is left blank. Thus programs that find a control sequence in **sshm** but none in **rshm** 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 **rshm**. 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. Note that the existing **msggr** 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. **msggr** should be set for those printers that accept any motion control sequences without affecting shadow, italicized, widened, superscript, or subscript printing. Conversely, if **msggr** is not set, a program should end these modes before attempting any motion.

Alternate Character Sets

In addition to allowing you to define line graphics, **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 scsd defc rcsd csnm daisy	Select character set N Start definition of character set N, M characters Define character A, B dots wide, descender D End definition of character set N List of character set names 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 **scsd** 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 **tparm**.

If a character set has to be defined before it can be used, the **scsd** 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 **tparm** 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**, **scsd**, and **rcsd** strings with a character set number in a call to **tparm**, a program can determine which of the three are needed.

Between use of the **scsd** 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 lower case 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.

It's 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 **tparm**, 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.

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, <i>N</i> , 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, <i>B</i> 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 a 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 an 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 lower case 'x' instead of a pin number, a 1 bit is used in the skipped position. For consistency, a lower case '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 **o,o,6,5,4,3,2,1;63**.

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

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 **orvi** 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.

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.

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.

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, 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**.

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.

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

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.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan*

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

*IBM Corporation
Site Counsel
2455 South Road*

Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMSdfp, JES2, JES3, and MVS™, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those

products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: [IBM Lifecycle Support for z/OS \(www.ibm.com/software/support/systemsz/lifecycle\)](http://www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at [Copyright and Trademark information \(www.ibm.com/legal/copytrade.shtml\)](http://www.ibm.com/legal/copytrade.shtml).

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.

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 the 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

A

add_wch interface for enhanced curses [14](#)
add_wchnstr interface for enhanced curses [15](#)
addch interface for curses [11](#)
addchstr interface for curses [11](#)
addnstr interface for enhanced curses [12](#)
addnwstr interface for enhanced curses [13](#)
alternate character sets [172](#)
application usage [176](#)
area clears [156](#)
attr_get interface for enhanced curses [16](#)
attroff interface for curses [16](#)

B

basic capabilities [154](#)
baudrate interface for curses [18](#)
beep interface for curses [18](#)
bkgd interface for enhanced curses [19](#)
bkgrnd interface for enhanced curses [20](#)
border interface for enhanced curses [21](#)
border_set curses for enhanced curses [22](#)
box interface for curses [23](#)
box_set interface for enhanced curses [23](#)
buffer size [175](#)

C

can_change_color interface for enhanced curses [24](#)
capabilities that cause movement [168](#)
cbreak interface for curses [26](#)
chgat interface for enhanced curses [27](#)
clear interface for curses [27](#)
clearok interface for curses [28](#)
clrtobot interface for curses [29](#)
clrtoeol interface for curses [30](#)
color manipulation [163](#)
color_content interface for enhanced curses [30](#)
color_pairs interface for enhanced curses [31](#)
cols interface for enhanced curses [31](#)
conventions for device aliases [176](#)
copywin interface for curses [31](#)
cur_term interface for enhanced curses [33](#)
current window structure [7](#)
curs_set interface for enhanced curses [32](#)
curscr interface for curses [32](#)
curses environment, windows [7](#)
curses interfaces [11](#)
curses library [1](#)
curses.h header [121](#)
cursor motions [156](#)

D

def_prog_mode interface for curses [33](#)

default window structure [7](#)
defined capabilities [139](#)
del_curterm interface for enhanced curses [35](#)
delay_output interface for curses [34](#)
delays [161](#)
delch interface for curses [35](#)
deleteln interface for curses [37](#)
delscreen interface for curses [37](#)
delwin interface for curses [38](#)
derwin interface for curses [38](#)
device capabilities [154](#)
dot-matrix graphics [173](#)
doupdate interface for curses [39](#)
dupwin interface for enhanced curses [40](#)

E

echo interface for curses [40](#)
echo_wchar interface for enhanced curses [42](#)
echochar interface for enhanced curses [41](#)
effect of changing printing resolution [174](#)
endwin interface for curses [42](#)
erase interface for curses [43](#)
erasechar interface for curses [43](#)

F

filter interface for enhanced curses [44](#)
flash interface for curses [45](#)
flushinp interface for curses [45](#)
formal grammar [138](#)

G

get_wch interface for enhanced curses [52](#)
get_wstr interface for enhanced curses [54](#)
getbegyx interface for curses [46](#)
getbkgd interface for enhanced curses [47](#)
getbkgrnd interface for enhanced curses [47](#)
getcchar interface for enhanced curses [47](#)
getch interface for curses [48](#)
getmaxyx interface for enhanced curses [49](#)
getn_wstr interface for enhanced curses [51](#)
getnstr interface for curses [50](#)
getparyx interface for enhanced curses [52](#)
getstr interface for curses [52](#)
getwin interface for enhanced curses [53](#)
getyx interface for curses [54](#)

H

halfdelay interface for enhanced curses [55](#)
has_colors interface for enhanced curses [55](#)
has_ic interface for curses [55](#)
headers [121](#)
highlighting [158](#)

hline interface for enhanced curses [56](#), [57](#)
hline_set interface for enhanced curses [57](#), [58](#)

I

idcok interface for enhanced curses [58](#)
idllok interface for curses [59](#)
immedok interface for enhanced curses [59](#)
in_wch interface for enhanced curses [71](#)
in_wchnstr interface for enhanced curses [71](#)
inch interface for curses [59](#)
inchnstr interface for enhanced curses [60](#)
init_color interface for enhanced curses [61](#)
initializing curses [5](#)
initscr interface for curses [61](#), [62](#)
innstr interface for enhanced curses [62](#)
innwstr interface for enhanced curses [64](#)
ins_nwstr interface for enhanced curses [68](#)
ins_wch interface for enhanced curses [69](#)
ins_wstr interface for enhanced curses [70](#)
insch interface for curses [65](#)
insdeln interface for enhanced curses [65](#)
insert/delete character [157](#)
insert/delete line [157](#)
insertln interface for curses [66](#)
insnstr interface for enhanced curses [67](#)
insstr interface for enhanced curses [68](#)
instr interface for enhanced curses [69](#)
intrflush interface for curses [70](#)
inwstr interface for enhanced curses [72](#)
is_linetouched interface for curses [73](#)
isendwin interface for enhanced curses [72](#)

K

keyname interface for curses [74](#)
keypad [160](#)
keypad interface for curses [75](#)
killchar interface for curses [75](#)

L

leaveok interface for curses [76](#)
line graphics [162](#)
lines interface for enhanced curses [76](#)
longname interface for curses [76](#)

M

mainframe
 education [xxvi](#)
manipulating window data [9](#)
meta interface for enhanced curses [77](#)
minimum guaranteed limits [138](#)
miscellaneous [165](#)
move interface for curses [78](#)
mv interface for curses [78](#)
mvcur interface for enhanced curses [80](#)
mvderwin interface for enhanced curses [81](#)
mvprintw interface for curses [81](#)
mvscanw interface for curses [82](#)
mvwin interface for curses [82](#)

N

naming conventions [2](#)
napms interface for curses [83](#)
newpad interface for curses [84](#)
newterm interface for curses [85](#)
newwin interface for curses [85](#)
nl interface for curses [85](#)
no interface for curses [86](#)
nodelay interface for curses [86](#)
noqiflush interface for enhanced curses [87](#)
notimeout interface for enhanced curses [88](#)

O

overlay interface for curses [88](#)

P

pads [8](#)
pair_content interface for enhanced curses [89](#)
parameterized strings [155](#)
pechochar interface for enhanced curses [89](#)
pnoutrefresh interface for curses [90](#)
print quality [175](#)
printer capabilities [166](#)
printer resolution [166](#)
printing rate [175](#)
printwr interface for curses [90](#)
putp interface for enhanced curses [91](#)
putwin interface for enhanced curses [92](#)

Q

qiflush interface for enhanced curses [92](#)

R

raw interface for curses [92](#)
redrawwin interface for enhanced curses [93](#)
refresh interface for curses [93](#)
reset_prog_mode interface for curses [94](#)
resetty interface for curses [94](#)
restartterm interface for enhanced curses [94](#)
ripline interface for enhanced curses [95](#)
rounding values [166](#)

S

sample entry [152](#)
savetty interface for curses [96](#)
scanw interface for curses [96](#)
scr_dump interface for enhanced curses [96](#)
sclr interface for curses [97](#)
scrllok interface for curses [98](#)
selecting a terminal [175](#)
set_curterm interface for enhanced curses [99](#)
set_term interface for curses [99](#)
setccar interface for enhanced curses [98](#)
setscrreg interface for curses [99](#)
setupterm interface for enhanced curses [100](#)
similar terminals [166](#)
slk_attroff interface for enhanced curses [100](#)

- source file syntax [137](#)
- special cases [165](#)
- specifying printer resolution [167](#)
- standend interface for curses [102](#)
- start_color interface for enhanced curses [103](#)
- status lines [161](#)
- stdscr interface for enhanced cursor [103](#)
- structure of a curses program [3](#)
- subpad interface for enhanced curses [103](#)
- subwin interface for curses [104](#)
- subwindows [8](#)
- syncok interface for enhanced curses [104](#)

T

- tabs and initialization [160](#)
- term.h header for enhanced curses [134](#)
- termattrs interface for enhanced curses [105](#)
- terminfo source format [137](#)
- terminology [1](#)
- termname interface for enhanced curses [105](#)
- tgetent interface for enhanced curses [106](#)
- tigetflag interface for enhanced curses [107](#)
- timeout interface for enhanced curses [108](#)
- touchline interface for curses [108](#)
- tparm interface for enhanced curses [108](#)
- tputs interface for enhanced curses [109](#)
- trademarks [182](#)
- typeahead interface for enhanced curses [109](#)
- types of capabilities in the sample entry [152](#)

U

- unctrl header [135](#)
- unctrl interface for curses [110](#)
- underlining [158](#)
- ungetch interface for enhanced curses [110](#)
- untouchwin interface for enhanced curses [111](#)
- use_env interface for enhanced curses [111](#)

V

- variations of terminal definitions [176](#)
- vidattr interface for enhanced curses [112](#)
- visible bells [158](#)
- vline interface for enhanced curses [113](#)
- vline_set interface for enhanced curses [113](#)
- vw_printw interface for enhanced curses [114](#)
- vw_scanw interface for enhanced curses [115](#)
- vwprintw interface for enhanced curses [113](#)
- vwscanw interface for enhanced curses [115](#)

W

- w interface for curses [116](#)
- wunctrl interface for enhanced curses [118](#)

Z

- z/OS Basic Skills Documentation [xxvi](#)



Product Number: 5650-ZOS

SA38-0690-50

