A Hacker's Guide to NCURSES

Contents

Abstract

This document is a hacker's tour of the ncurses library and utilities.
It discusses design philosophy, implementation methods, and the
conventions used for coding and documentation. It is recommended
reading for anyone who is interested in porting, extending or
improving the package.

Objective of the Package

The objective of the ncurses package is to provide a free software API
for character-cell terminals and terminal emulators with the following
characteristics:
  * Source-compatible   with   historical   curses   implementations
    (including the original BSD curses and System V curses.
  * Conformant with the XSI Curses standard issued as part of XPG4 by
    X/Open.
  * High-quality -- stable and reliable code, wide portability, good
    packaging, superior documentation.
  * Featureful -- should eliminate as much of the drudgery of C
    interface programming as possible, freeing programmers to think at
    a higher level of design.

These objectives are in priority order. So, for example, source
compatibility with older version must trump featurefulness -- we
cannot add features if it means breaking the portion of the API
corresponding to historical curses versions.

Why System V Curses?

We used System V curses as a model, reverse-engineering their API, in
order to fulfill the first two objectives.

System V curses implementations can support BSD curses programs with

just a recompilation, so by capturing the System V API we also capture
BSD's.

More importantly for the future, the XSI Curses standard issued by
X/Open is explicitly and closely modeled on System V. So conformance
with System V took us most of the way to base-level XSI conformance.

How to Design Extensions

The third objective (standards conformance) requires that it be easy
to condition source code using ncurses so that the absence of
nonstandard extensions does not break the code.

Accordingly, we have a policy of associating with each nonstandard
extension a feature macro, so that ncurses client code can use this
macro to condition in or out the code that requires the ncurses
extension.

For example, there is a macro NCURSES_MOUSE_VERSION which XSI Curses
does not define, but which is defined in the ncurses library header.
You can use this to condition the calls to the mouse API calls.

                        Portability and Configuration

Code written for ncurses may assume an ANSI-standard C compiler and
POSIX-compatible OS interface. It may also assume the presence of a
System-V-compatible select(2) call.

We encourage (but do not require) developers to make the code friendly
to less-capable UNIX environments wherever possible.

We encourage developers to support OS-specific optimizations and
methods not available under POSIX/ANSI, provided only that:
   * All such code is properly conditioned so the build process does
     not attempt to compile it under a plain ANSI/POSIX environment.
   * Adding such implementation methods does not introduce
     incompatibilities in the ncurses API between platforms.

We use GNU autoconf(1) as a tool to deal with portability issues. The
right way to leverage an OS-specific feature is to modify the autoconf
specification files (configure.in and aclocal.m4) to set up a new
feature macro, which you then use to condition your code.

                        Documentation Conventions

There are three kinds of documentation associated with this package.
Each has a different preferred format:
   * Package-internal files (README, INSTALL, TO-DO etc.)
   * Manual pages.
   * Everything else (i.e., narrative documentation).

Our conventions are simple:
  1. Maintain package-internal files in plain text. The expected viewer
     for them more(1) or an editor window; there is no point in
     elaborate mark-up.
  2. Mark up manual pages in the man macros. These have to be viewable
     through traditional man(1) programs.
  3. Write everything else in HTML.

When in doubt, HTMLize a master and use lynx(1) to generate plain

ASCII (as we do for the announcement document).

The reason for choosing HTML is that it is (a) well-adapted for
on-line browsing through viewers that are everywhere; (b) more easily
readable as plain text than most other mark-ups, if you do not have a
viewer; and (c) carries enough information that you can generate a
nice-looking printed version from it. Also, of course, it make
exporting things like the announcement document to WWW pretty trivial.

How to Report Bugs

The reporting address for bugs is bug-ncurses@gnu.org. This is a
majordomo list; to join, write to bug-ncurses-request@gnu.org with a
message containing the line:
        subscribe <name>@<host.domain>

The ncurses code is maintained by a small group of volunteers. While
we try our best to fix bugs promptly, we simply do not have a lot of
hours to spend on elementary hand-holding. We rely on intelligent
cooperation from our users. If you think you have found a bug in
ncurses, there are some steps you can take before contacting us that
will help get the bug fixed quickly.

In order to use our bug-fixing time efficiently, we put people who
show us they have taken these steps at the head of our queue. This
means that if you do not, you will probably end up at the tail end and
have to wait a while.
  1. Develop a recipe to reproduce the bug.
     Bugs we can reproduce are likely to be fixed very quickly, often
     within days. The most effective single thing you can do to get a
     quick fix is develop a way we can duplicate the bad behavior --
     ideally, by giving us source for a small, portable test program
     that breaks the library. (Even better is a keystroke recipe using
     one of the test programs provided with the distribution.)
  2. Try to reproduce the bug on a different terminal type.
     In our experience, most of the behaviors people report as library
     bugs are actually due to subtle problems in terminal descriptions.
     This is especially likely to be true if you are using a
     traditional asynchronous terminal or PC-based terminal emulator,
     rather than xterm or a UNIX console entry.
     It is therefore extremely helpful if you can tell us whether or
     not your problem reproduces on other terminal types. Usually you
     will have both a console type and xterm available; please tell us
     whether or not your bug reproduces on both.
     If you have xterm available, it is also good to collect xterm
     reports for different window sizes. This is especially true if you
     normally use an unusual xterm window size -- a surprising number
     of the bugs we have seen are either triggered or masked by these.
  3. Generate and examine a trace file for the broken behavior.
     Recompile your program with the debugging versions of the
     libraries. Insert a trace() call with the argument set to
     TRACE_UPDATE. (See "Writing Programs with NCURSES" for details on
     trace levels.) Reproduce your bug, then look at the trace file to
     see what the library was actually doing.
     Another frequent cause of apparent bugs is application coding
     errors that cause the wrong things to be put on the virtual
     screen. Looking at the virtual-screen dumps in the trace file will
     tell you immediately if this is happening, and save you from the
     possible embarrassment of being told that the bug is in your code
     and is your problem rather than ours.

If the virtual−screen dumps look correct but the bug persists, it is possible to crank up the trace level to give more and more information about the library's update actions and the control sequences it issues to perform them. The test directory of the distribution contains a tool for digesting these logs to make them less tedious to wade through.

Often you will find terminfo problems at this stage by noticing that the escape sequences put out for various capabilities are wrong. If not, you are likely to learn enough to be able to characterize any bug in the screen−update logic quite exactly.

4. Report details and symptoms, not just interpretations.

If you do the preceding two steps, it is very likely that you will discover the nature of the problem yourself and be able to send us a fix. This will create happy feelings all around and earn you good karma for the first time you run into a bug you really cannot characterize and fix yourself.

If you are still stuck, at least you will know what to tell us. Remember, we need details. If you guess about what is safe to leave out, you are too likely to be wrong.

If your bug produces a bad update, include a trace file. Try to make the trace at the least voluminous level that pins down the bug. Logs that have been through tracemunch are OK, it does not throw away any information (actually they are better than un−munched ones because they are easier to read).

If your bug produces a core−dump, please include a symbolic stack trace generated by gdb(1) or your local equivalent.

Tell us about every terminal on which you have reproduced the bug −− and every terminal on which you cannot. Ideally, send us terminfo sources for all of these (yours might differ from ours). Include your ncurses version and your OS/machine type, of course! You can find your ncurses version in the curses.h file.

If your problem smells like a logic error or in cursor movement or scrolling or a bad capability, there are a couple of tiny test frames for the library algorithms in the progs directory that may help you isolate it. These are not part of the normal build, but do have their own make productions.

The most important of these is mvcur, a test frame for the cursor−movement optimization code. With this program, you can see directly what control sequences will be emitted for any given cursor movement or scroll/insert/delete operations. If you think you have got a bad capability identified, you can disable it and test again. The program is command−driven and has on−line help.

If you think the vertical−scroll optimization is broken, or just want to understand how it works better, build hashmap and read the header comments of hardscroll.c and hashmap.c; then try it out. You can also test the hardware−scrolling optimization separately with hardscroll.

A Tour of the Ncurses Library

Library Overview

Most of the library is superstructure −− fairly trivial convenience interfaces to a small set of basic functions and data structures used to manipulate the virtual screen (in particular, none of this code does any I/O except through calls to more fundamental modules described below). The files

```
lib_addch.c    lib_bkgd.c   lib_box.c    lib_chgat.c   lib_clear.c
lib_clearok.c  lib_clrbot.c lib_clreol.c lib_colorset.c lib_data.c
lib_delch.c    lib_delwin.c  lib_echo.c   lib_erase.c   lib_gen.c
lib_getstr.c lib_hline.c lib_immedok.c lib_inchstr.c lib_insch.c
lib_insdel.c  lib_insstr.c lib_instr.c lib_isendwin.c lib_keyname.c
lib_leaveok.c  lib_move.c   lib_mvwin.c   lib_overlay.c lib_pad.c
lib_printw.c lib_redrawln.c  lib_scanw.c lib_screen.c lib_scroll.c
lib_scrollok.c      lib_scrreg.c      lib_set_term.c      lib_slk.c
lib_slkatr_set.c  lib_slkatrof.c  lib_slkatron.c lib_slkatrset.c
lib_slkattr.c     lib_slkclear.c    lib_slkcolor.c   lib_slkinit.c
lib_slklab.c  lib_slkrefr.c lib_slkset.c lib_slktouch.c lib_touch.c
lib_unctrl.c lib_vline.c lib_wattroff.c lib_wattron.c lib_window.c
```

are all in this category. They are very unlikely to need change,
barring bugs or some fundamental reorganization in the underlying data
structures.

These files are used only for debugging support:

```
lib_trace.c     lib_traceatr.c    lib_tracebits.c    lib_tracechr.c
lib_tracedmp.c lib_tracemse.c trace_buf.c
```

It is rather unlikely you will ever need to change these, unless you
want to introduce a new debug trace level for some reason.

There is another group of files that do direct I/O via tputs(),
computations on the terminal capabilities, or queries to the OS
environment, but nevertheless have only fairly low complexity. These
include:

```
lib_acs.c  lib_beep.c  lib_color.c  lib_endwin.c   lib_initscr.c
lib_longname.c  lib_newterm.c  lib_options.c lib_termcap.c lib_ti.c
lib_tparm.c lib_tputs.c lib_vidattr.c read_entry.c.
```

They are likely to need revision only if ncurses is being ported to an
environment without an underlying terminfo capability representation.

These files have serious hooks into the tty driver and signal
facilities:

```
lib_kernel.c lib_baudrate.c lib_raw.c lib_tstp.c lib_twait.c
```

If you run into porting snafus moving the package to another UNIX, the
problem is likely to be in one of these files. The file lib_print.c
uses sleep(2) and also falls in this category.

Almost all of the real work is done in the files

```
hardscroll.c   hashmap.c   lib_addch.c  lib_doupdate.c  lib_getch.c
lib_mouse.c lib_mvcur.c lib_refresh.c lib_setup.c lib_vidattr.c
```

Most of the algorithmic complexity in the library lives in these
files. If there is a real bug in ncurses itself, it is probably here.
We will tour some of these files in detail below (see The Engine
Room).

Finally, there is a group of files that is actually most of the
terminfo compiler. The reason this code lives in the ncurses library
is to support fallback to /etc/termcap. These files include

```
      alloc_entry.c captoinfo.c comp_captab.c comp_error.c comp_hash.c
      comp_parse.c comp_scan.c parse_entry.c read_termcap.c write_entry.c
```

   We will discuss these in the compiler tour.

The Engine Room

  Keyboard Input

   All ncurses input funnels through the function wgetch(), defined in
   lib_getch.c. This function is tricky; it has to poll for keyboard and
   mouse events and do a running match of incoming input against the set
   of defined special keys.

   The central data structure in this module is a FIFO queue, used to
   match   multiple-character   input   sequences   against   special-key
   capabilities; also to implement pushback via ungetch().

   The wgetch() code distinguishes between function key sequences and the
   same sequences typed manually by doing a timed wait after each input
   character that could lead a function key sequence. If the entire
   sequence takes less than 1 second, it is assumed to have been
   generated by a function key press.

   Hackers bruised by previous encounters with variant select(2) calls
   may find the code in lib_twait.c interesting. It deals with the
   problem that some BSD selects do not return a reliable time-left
   value. The function timed_wait() effectively simulates a System V
   select.

  Mouse Events

   If the mouse interface is active, wgetch() polls for mouse events each
   call, before it goes to the keyboard for input. It is up to
   lib_mouse.c how the polling is accomplished; it may vary for different
   devices.

   Under xterm, however, mouse event notifications come in via the
   keyboard input stream. They are recognized by having the kmous
   capability as a prefix. This is kind of klugey, but trying to wire in
   recognition of a mouse key prefix without going through the
   function-key machinery would be just too painful, and this turns out
   to imply having the prefix somewhere in the function-key capabilities
   at terminal-type initialization.

   This kluge only works because kmous is not actually used by any
   historic terminal type or curses implementation we know of. Best guess
   is it is a relic of some forgotten experiment in-house at Bell Labs
   that did not leave any traces in the publicly-distributed System V
   terminfo files. If System V or XPG4 ever gets serious about using it
   again, this kluge may have to change.

   Here are some more details about mouse event handling:

   The lib_mouse() code is logically split into a lower level that
   accepts event reports in a device-dependent format and an upper level
   that parses mouse gestures and filters events. The mediating data
   structure is a circular queue of event structures.

   Functionally, the lower level's job is to pick up primitive events and

put them on the circular queue. This can happen in one of two ways:
either (a) _nc_mouse_event() detects a series of incoming mouse
reports and queues them, or (b) code in lib_getch.c detects the kmous
prefix in the keyboard input stream and calls _nc_mouse_inline to
queue up a series of adjacent mouse reports.

In either case, _nc_mouse_parse() should be called after the series is
accepted to parse the digested mouse reports (low−level events) into a
gesture (a high−level or composite event).

Output and Screen Updating

With the single exception of character echoes during a wgetnstr() call
(which simulates cooked−mode line editing in an ncurses window), the
library normally does all its output at refresh time.

The main job is to go from the current state of the screen (as
represented in the curscr window structure) to the desired new state
(as represented in the newscr window structure), while doing as little
I/O as possible.

The brains of this operation are the modules hashmap.c, hardscroll.c
and lib_doupdate.c; the latter two use lib_mvcur.c. Essentially, what
happens looks like this:
   * The hashmap.c module tries to detect vertical motion changes
     between the real and virtual screens. This information is
     represented by the oldindex members in the newscr structure. These
     are modified by vertical−motion and clear operations, and both are
     re−initialized after each update. To this change−journalling
     information, the hashmap code adds deductions made using a
     modified Heckel algorithm on hash values generated from the line
     contents.
   * The hardscroll.c module computes an optimum set of scroll,
     insertion, and deletion operations to make the indices match. It
     calls _nc_mvcur_scrolln() in lib_mvcur.c to do those motions.
   * Then lib_doupdate.c goes to work. Its job is to do line−by−line
     transformations of curscr lines to newscr lines. Its main tool is
     the routine mvcur() in lib_mvcur.c. This routine does
     cursor−movement optimization, attempting to get from given screen
     location A to given location B in the fewest output characters
     possible.

If you want to work on screen optimizations, you should use the fact
that (in the trace−enabled version of the library) enabling the
TRACE_TIMES trace level causes a report to be emitted after each
screen update giving the elapsed time and a count of characters
emitted during the update. You can use this to tell when an update
optimization improves efficiency.

In the trace−enabled version of the library, it is also possible to
disable and re−enable various optimizations at runtime by tweaking the
variable _nc_optimize_enable. See the file include/curses.h.in for
mask values, near the end.

                    The Forms and Menu Libraries

The forms and menu libraries should work reliably in any environment
you can port ncurses to. The only portability issue anywhere in them
is what flavor of regular expressions the built−in form field type
TYPE_REGEXP will recognize.

The configuration code prefers the POSIX regex facility, modeled on
System V's, but will settle for BSD regexps if the former is not
available.

Historical note: the panels code was written primarily to assist in
porting u386mon 2.0 (comp.sources.misc v14i001-4) to systems lacking
panels support; u386mon 2.10 and beyond use it. This version has been
slightly cleaned up for ncurses.

                         A Tour of the Terminfo Compiler

The ncurses implementation of tic is rather complex internally; it has
to do a trying combination of missions. This starts with the fact
that, in addition to its normal duty of compiling terminfo sources
into loadable terminfo binaries, it has to be able to handle termcap
syntax and compile that too into terminfo entries.

The implementation therefore starts with a table-driven, dual-mode
lexical analyzer (in comp_scan.c). The lexer chooses its mode (termcap
or terminfo) based on the first "," or ":" it finds in each entry. The
lexer does all the work of recognizing capability names and values;
the grammar above it is trivial, just "parse entries till you run out
of file".

Translation of Non-use Capabilities

Translation of most things besides use capabilities is pretty
straightforward. The lexical analyzer's tokenizer hands each
capability name to a hash function, which drives a table lookup. The
table entry yields an index which is used to look up the token type in
another table, and controls interpretation of the value.

One possibly interesting aspect of the implementation is the way the
compiler tables are initialized. All the tables are generated by
various awk/sed/sh scripts from a master table include/Caps; these
scripts actually write C initializers which are linked to the
compiler. Furthermore, the hash table is generated in the same way, so
it doesn't have to be generated at compiler startup time (another
benefit of this organization is that the hash table can be in
shareable text space).

Thus, adding a new capability is usually pretty trivial, just a matter
of adding one line to the include/Caps file. We will have more to say
about this in the section on Source-Form Translation.

Use Capability Resolution

The background problem that makes tic tricky is not the capability
translation itself, it is the resolution of use capabilities. Older
versions would not handle forward use references for this reason (that
is, a using terminal always had to follow its use target in the source
file). By doing this, they got away with a simple implementation
tactic; compile everything as it blows by, then resolve uses from
compiled entries.

This will not do for ncurses. The problem is that that the whole
compilation process has to be embeddable in the ncurses library so
that it can be called by the startup code to translate termcap entries
on the fly. The embedded version cannot go promiscuously writing

everything it translates out to disk -- for one thing, it will
typically be running with non-root permissions.

So our tic is designed to parse an entire terminfo file into a
doubly-linked circular list of entry structures in-core, and then do
use resolution in-memory before writing everything out. This design
has other advantages: it makes forward and back use-references equally
easy (so we get the latter for free), and it makes checking for name
collisions before they are written out easy to do.

And this is exactly how the embedded version works. But the
stand-alone user-accessible version of tic partly reverts to the
historical strategy; it writes to disk (not keeping in core) any entry
with no use references.

This is strictly a core-economy kluge, implemented because the
terminfo master file is large enough that some core-poor systems swap
like crazy when you compile it all in memory...there have been reports
of this process taking three hours, rather than the twenty seconds or
less typical on the author's development box.

So. The executable tic passes the entry-parser a hook that immediately
writes out the referenced entry if it has no use capabilities. The
compiler main loop refrains from adding the entry to the in-core list
when this hook fires. If some other entry later needs to reference an
entry that got written immediately, that is OK; the resolution code
will fetch it off disk when it cannot find it in core.

Name collisions will still be detected, just not as cleanly. The
write_entry() code complains before overwriting an entry that
postdates the time of tic's first call to write_entry(), Thus it will
complain about overwriting entries newly made during the tic run, but
not about overwriting ones that predate it.

Source-Form Translation

Another use of tic is to do source translation between various termcap
and terminfo formats. There are more variants out there than you might
think; the ones we know about are described in the captoinfo(1) manual
page.

The translation output code (dump_entry() in ncurses/dump_entry.c) is
shared with the infocmp(1) utility. It takes the same internal
representation used to generate the binary form and dumps it to
standard output in a specified format.

The include/Caps file has a header comment describing ways you can
specify source translations for nonstandard capabilities just by
altering the master table. It is possible to set up capability
aliasing or tell the compiler to plain ignore a given capability
without writing any C code at all.

For circumstances where you need to do algorithmic translation, there
are functions in parse_entry.c called after the parse of each entry
that are specifically intended to encapsulate such translations. This,
for example, is where the AIX box1 capability get translated to an
acsc string.

                              Other Utilities

The infocmp utility is just a wrapper around the same entry-dumping
code used by tic for source translation. Perhaps the one interesting
aspect of the code is the use of a predicate function passed in to
dump_entry() to control which capabilities are dumped. This is
necessary in order to handle both the ordinary De-compilation case and
entry difference reporting.

The tput and clear utilities just do an entry load followed by a
tputs() of a selected capability.

<div align="center">Style Tips for Developers</div>

See the TO-DO file in the top-level directory of the source
distribution for additions that would be particularly useful.

The prefix _nc_ should be used on library public functions that are
not part of the curses API in order to prevent pollution of the
application namespace. If you have to add to or modify the function
prototypes in curses.h.in, read ncurses/MKlib_gen.sh first so you can
avoid breaking XSI conformance. Please join the ncurses mailing list.
See the INSTALL file in the top level of the distribution for details
on the list.

Look for the string FIXME in source files to tag minor bugs and
potential problems that could use fixing.

Do not try to auto-detect OS features in the main body of the C code.
That is the job of the configuration system.

To hold down complexity, do make your code data-driven. Especially, if
you can drive logic from a table filtered out of include/Caps, do it.
If you find you need to augment the data in that file in order to
generate the proper table, that is still preferable to ad-hoc code --
that is why the fifth field (flags) is there.

Have fun!

<div align="center">Porting Hints</div>

The following notes are intended to be a first step towards DOS and
Macintosh ports of the ncurses libraries.

The following library modules are "pure curses"; they operate only on
the curses internal structures, do all output through other curses
calls (not including tputs() and putp()) and do not call any other
UNIX routines such as signal(2) or the stdio library. Thus, they
should not need to be modified for single-terminal ports.

```
  lib_addch.c    lib_addstr.c    lib_bkgd.c   lib_box.c   lib_clear.c
  lib_clrbot.c   lib_clreol.c   lib_delch.c   lib_delwin.c  lib_erase.c
  lib_inchstr.c  lib_insch.c   lib_insdel.c  lib_insstr.c  lib_keyname.c
  lib_move.c   lib_mvwin.c   lib_newwin.c   lib_overlay.c   lib_pad.c
  lib_printw.c  lib_refresh.c  lib_scanw.c   lib_scroll.c  lib_scrreg.c
  lib_set_term.c  lib_touch.c  lib_tparm.c  lib_tputs.c  lib_unctrl.c
  lib_window.c  panel.c
```

This module is pure curses, but calls outstr():

```
  lib_getstr.c
```

These modules are pure curses, except that they use tputs() and
putp():

  lib_beep.c lib_color.c lib_endwin.c lib_options.c lib_slk.c
  lib_vidattr.c

This modules assist in POSIX emulation on non−POSIX systems:

sigaction.c
      signal calls

The following source files will not be needed for a
single-terminal−type port.

  alloc_entry.c captoinfo.c clear.c comp_captab.c comp_error.c
  comp_hash.c comp_main.c comp_parse.c comp_scan.c dump_entry.c
  infocmp.c parse_entry.c read_entry.c tput.c write_entry.c

The following modules will use open()/read()/write()/close()/lseek()
on files, but no other OS calls.

lib_screen.c
      used to read/write screen dumps

lib_trace.c
      used to write trace data to the logfile

Modules that would have to be modified for a port start here:

The following modules are "pure curses" but contain assumptions
inappropriate for a memory−mapped port.

lib_longname.c
      assumes there may be multiple terminals

lib_acs.c
      assumes acs_map as a double indirection

lib_mvcur.c
      assumes cursor moves have variable cost

lib_termcap.c
      assumes there may be multiple terminals

lib_ti.c
      assumes there may be multiple terminals

The following modules use UNIX−specific calls:

lib_doupdate.c
      input checking

lib_getch.c
      read()

lib_initscr.c
      getenv()

lib_newterm.c
lib_baudrate.c

```
lib_kernel.c
      various tty-manipulation and system calls

lib_raw.c
      various tty-manipulation calls

lib_setup.c
      various tty-manipulation calls

lib_restart.c
      various tty-manipulation calls

lib_tstp.c
      signal-manipulation calls

lib_twait.c
      gettimeofday(), select().
```
_____

 Eric S. Raymond <esr@snark.thyrsus.com>

(Note: This is not the bug address!)