1. Terminal/Host Interchange. This mdule is responsible for communicating with the host computer and interpreting the control sequences and graphic characters it sends. It begins with the C preamble followed by bookkeeping routines to initialise and destroy a terminal object. Following this there is a section of unix-specific code for managing signals and processes that ought to be moved elsewhere.

This is followed by the process for interpreting the byte stream from the host into control sequences, then an implementation of each operation that a control sequence can perform.

The header file created for this module is named `triterm.h` to avoid a clash with `term.h` from the Curses library.

```
⟨triterm.h 1⟩ ≡
#ifndef TURTLE_TERM_H
#define TURTLE_TERM_H
#include <limits.h>
#include <stddef.h>
#include <stdint.h>
#include <event2/event.h>
#include "buf.h"
#include "panel.h"
  ⟨Public API (term.o) 5⟩
#endif    /* TURTLE_TERM_H */
```

2. The C source begins by including header files and declaring globals.

```
#include "triterm.h"
#include <assert.h>
#include <err.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include "parsnip/safe-math.h"
#include "site.h"    /* common C junk */
#include "unicode.h"    /* UTF-8 */
  ⟨Type definitions (term.o) 4⟩
  ⟨Global variables (term.o) 18⟩
  ⟨Function declarations (term.o) 9⟩
```

3. These headers are only required for the mis-placed unix specialisation.

```
#include <pwd.h>
#include <signal.h>
#include <sys/ioctl.h>    /* non-standard */
#include <sys/wait.h>
#include <termios.h>
#include <unistd.h>
#ifdef __GLIBC__
#include <pty.h>
#elif __FreeBSD__
#include <libutil.h>
#else
#include <util.h>
#endif
```

4. This object references all of the run-time state of a single terminal instance; its contents will be described as they are used. I don't think this is the right place to introduce these flag values.

```
#define TERM_AUTO_WRAP 0
#define TERM_CURSOR_VISIBLE 1
#define TERM_EIGHTBIT 2
#define TERM_INSERT_REPLACE 3
#define TERM_IS_LIVE 4
#define TERM_LAST_COLUMN 5
#define TERM_LEFT_RIGHT_MARGIN 6
#define TERM_NEW_LINE_MODE 7
#define TERM_ORIGIN_MOTION 8
#define TERM_PARSER_ACTIVE 9
#define TERM_UTF_HOST 10

#define TERM_BUFSIZE 1024
#define TERM_MAXSIZE (1048576 * 16)     /* 16MB */
```

⟨Type definitions (term.o) 4⟩ ≡

```
  ⟨Definition of gset_xt object 17⟩
  typedef struct term_xt {
    struct event_base *loop;     /* Event loop this panel will join. */
    struct event *ev_signal;
    struct event *ev_io;
    pid_t child_pid;
    int child_fd;
    int exit_status;
    int flags;
    term_act_fun bell;     /* alert (bell) action */
    term_act_fun billdoor;     /* Reaper Man; RIP PTERRY */
    panel_xt *panel;     /* should be in-line */
    void *resume;     /* parser return state */
    utfio_xt utf;     /* wastes four bytes */
    uint8_t gr96[4];     /* bool */
    int use_gl, use_gr;     /* 0, 1, 2, 3 (0 cannot go in gr) */
    gset_xt *grachar[4];     /* see also TERM_UTF_HOST */
    size_t at;     /* parser offset in buf */
    uint8_t buf[TERM_BUFSIZE];     /* buffer consume reads from */
    action_xt parser;     /* result of parsing a control sequence */
    struct timeval slow_time;     /* simulating a slow baud rate (badly) */
    struct event *slow_more;
    float slow, slow_budget, slow_tick;
    int lio_dir;     /* tracing communication with the host */
    size_t lio_len;
    FILE *lio_fs;
  } term_xt;
```

See also section 149.

This code is used in section 2.

5. Input from the host is parsed into values in this object which is then used by the control operators. It can also be filled in by the user to call the control operators directly so it is made public.

```
⟨ Public API (term.o) 5 ⟩ ≡
#define TERM_ARG_SIZE 16
#define TERM_ARG_LIMIT INT_MAX
  typedef struct term_xt term_xt;
  typedef struct action_xt {
    buf_xt data;
    int16_t flags;
    uint8_t mode;
    uint8_t prefix[2];
    uint8_t reply;
    int8_t _pad;
    int8_t narg;
    int arg[TERM_ARG_SIZE];
  } action_xt;
```

See also sections 6, 7, 8, 20, 31, 59, 140, 142, and 150.

This code is used in section 1.

6. The user can register functions that perform certain terminal actions, eg. to ring the `BEL`.

```
⟨ Public API (term.o) 5 ⟩ +≡
  typedef void(∗term_act_fun)(term_xt ∗);
```

7. Here is the public API for initialisation and destruction of a terminal.

```
⟨ Public API (term.o) 5 ⟩ +≡
  term_xt ∗new_term(struct event_base ∗, panel_xt ∗);
  void term_destroy(term_xt ∗);
  void term_set_bell_fun(term_xt ∗, term_act_fun);
  void term_set_reap_fun(term_xt ∗, term_act_fun);
```

8. Speed feature.

```
⟨ Public API (term.o) 5 ⟩ +≡
  void term_set_speed(term_xt ∗, float, float);
```

9. And here is some internal functions that should move elsewhere.

```
⟨ Function declarations (term.o) 9 ⟩ ≡
  static void _term_read(int, short, void ∗);
  static void _term_slow_consume(term_xt ∗, size_t);
  static void _term_slow_more(int, short, void ∗);
```

See also sections 21, 32, 141, and 151.

This code is used in section 2.

10. Initialise a new terminal object.
TODO: Make tracing optional.
TODO: Panel should be initialised here rather than the caller.
TODO: rename *new_term*.

```
term_xt *new_term(struct event_base *loop, panel_xt *panel)
{
  term_xt *ret;
  ret ← calloc(1, sizeof(term_xt));
  if (¬ret) return Λ;
  ret→loop ← loop;
  ret→panel ← panel;    /* TODO: alloc/init here */
  if (¬buf_init(&ret→parser.data, 1, TERM_BUFSIZE, 0, TERM_MAXSIZE)) {
    free(ret);
    return Λ;
  }
  utfio_init(&ret→utf);
  CLEAR_FLAG(ret→flags, TERM_EIGHTBIT);
  SET_FLAG(ret→flags, TERM_IS_LIVE);
  SET_FLAG(ret→flags, TERM_AUTO_WRAP);
  SET_FLAG(ret→flags, TERM_UTF_HOST);
  ⟨Initialise 7-bit character sets 19⟩
  ret→child_fd ← −1;
  term_lio_start(ret, "/tmp/turtle.trace");
  return ret;
}
```

11. Destruction of the terminal object is more complicated than usual because there will usually be a process to wait for. If there is then this function clears the TERM_IS_LIVE flag to indicate that the caller no longer holds a reference to the terminal object. Cleanup will continue when the signal handler is called after the process exits.

```
void term_destroy(term_xt *tt)
{
  _lio_byte(tt, LIO_NONE, Λ, 0);
  CLEAR_FLAG(tt→flags, TERM_IS_LIVE);
  if (tt→child_fd ≠ −1) term_close_pty(tt);
  if (tt→child_pid) return;    /* not reaped yet */
  if (Handler_Active ≡ tt) term_remove_signal_handler(tt);
  free(tt);
}
```

12. These functions set the callbacks for the two bell-ringers.

```
void term_set_bell_fun(term_xt *tt, term_act_fun bell)
{
  tt→bell ← bell;
}
void term_set_reap_fun(term_xt *tt, term_act_fun reaper)
{
  tt→billdoor ← reaper;
}
```

13. When a pseudo-terminal is ready it will be associated with an event that calls this function when data are ready.

Turtle includes a feature to slow down the apparent rate when reading from the host to simulate the experience of connecting over a slow serial link. The implementation of this feature is not going to win any awards but it is effective: A "budget" of bytes is increased at the given rate and decreased whenever data are read, if the budget is zero then the function returns (TODO: libevent will call it again immediately: do it another way or remove the event from the loop until the budget is restored).

TODO: What does $r \equiv 0$ mean in this context (may not be a subprocess)?

```
void _term_read(int fd, short e, void *arg)
{
   term_xt *tt ← arg;
   ssize_t r;
   if (tt→slow ∧ tt→slow_budget ≤ 0) return;
   assert(e & EV_READ);
   assert(fd ≡ tt→child_fd);
again: r ← sizeof (tt→buf);
   if (tt→slow_budget < r) r ← tt→slow_budget;
   r ← read(fd, tt→buf, r);
   if (r ≡ −1) {
      if (errno ≡ EINTR) goto again;
      else {
         warn("read");
         return;
      }
   }
   if (r) _term_consume_bytes(tt, r);
   _term_slow_consume(tt, r);
}
```

14. To enable the slowdown a timer event is added running every *tick* seconds to increase the budget.

TODO: Add a flag or detect zero in *slow* better.

```
void term_set_speed(term_xt *tt, float bps, float tick)
{
   tt→slow_budget ← tt→slow ← bps;
   tt→slow_tick ← tick;
   tt→slow_time.tv_sec ← tick;
   tt→slow_time.tv_usec ← (tick − tt→slow_time.tv_sec) * 1000000;
   tt→slow_more ← evtimer_new(tt→loop, _term_slow_more, tt);
   evtimer_add(tt→slow_more, &tt→slow_time);
}
```

15.
```
static void _term_slow_more(int fd, short e, void *arg)
{
   term_xt *tt ← (term_xt *) arg;
   assert(fd ≡ −1);
   assert(e & EV_TIMEOUT);
   if (¬tt→slow) return;
   if (tt→slow_budget < tt→slow) tt→slow_budget += tt→slow * tt→slow_tick;
   evtimer_add(tt→slow_more, &tt→slow_time);
}
```

16.
```
static void _term_slow_consume(term_xt *tt, size_t got)
{
   tt→slow_budget −= got;
}
```

17. Character Sets. Multiple character sets are not supported in Turtle except ASCII and UTF-8-encoded Unicode but many applications expect it so a bare framework exists which can be expanded if necessary. Four character sets may be designated and one of these may then be invoked into GL (codes 0–127) or GR (code 128–255).

```
⟨ Definition of gset_xt object 17 ⟩ ≡
  typedef struct gset_xt {
    bool is_96;      /* sized to avoid noise about padding */
    ucp_xt codepoint[96];
  } gset_xt;
```

This code is used in section 4.

18. ⟨ Global variables (`term.o`) 18 ⟩ ≡

```
  gset_xt GTable_ASCII ← {0};
```

See also section 22.

This code is used in section 2.

19. DEC-STD-070 ss. 3.6.4. Level 1 has ASCII in g0-g1, g0 in gl. Does not expect to receive 8 bit anything, but of course we will so act like a vt220.

Level 2 w/o 8-bit has ASCII in g0/1 on gl and DEC Supplemental in g2/3 on gr.

Level 3 or 2 w/ has ASCII in g0/1 on gl, UPSS (DEC Supplemental or Latin-1) in g2/3 on gr.

Latter also recognises "Announce Subset Of Code Extension Facilities" in ISO 4873.

ASCII is replacable as a local option that is not implemented (NRCS).

```
⟨ Initialise 7-bit character sets 19 ⟩ ≡
  ret→use_gr ← 2;      /* Depends on conformance level. */
  ret→grachar[0] ← &GTable_ASCII;
  ret→grachar[1] ← &GTable_ASCII;
  ret→grachar[2] ← &GTable_ASCII;
  ret→grachar[3] ← &GTable_ASCII;
```

This code is used in section 10.

20. Unix. These parts are Unix only and should be moved somewhere else. They will be described later. For now the most interesting part is *term_attach_pty_exec* which initialises a new TTY.

```
⟨ Public API (term.o) 5 ⟩ +≡
  bool term_attach_pty_exec(term_xt ∗, char ∗, char ∗∗);
  void term_close_pty(term_xt ∗);
  bool term_has_pty(term_xt ∗);
  void term_install_signal_handler(term_xt ∗);
  void term_remove_signal_handler(term_xt ∗);
```

21. ⟨ Function declarations (term.o) 9 ⟩ +≡

```
  static void _term_install_io_handler(term_xt ∗);
  static void _term_reap(int, short, void ∗);
  static void _term_remove_io_handler(term_xt ∗);
```

22. ⟨ Global variables (term.o) 18 ⟩ +≡

```
  term_xt ∗Handler_Active ← Λ;
```

23.

```
void term_install_signal_handler(term_xt ∗tt)
  {
    struct sigaction old_sa;
    if (Handler_Active) {
    already: warn("SIGCHLD␣already␣handled");
      return;
    }
    if (sigaction(SIGCHLD, Λ, &old_sa) ≡ −1) {
      warn("sigaction");
      return;
    }
    if (old_sa.sa_handler ≠ SIG_DFL) goto already;
    tt→ev_signal ← evsignal_new(tt→loop, SIGCHLD, _term_reap, tt);
    evsignal_add(tt→ev_signal, Λ);
    Handler_Active ← tt;
  }
```

24. TODO: Also check the handler?

```
  void term_remove_signal_handler(term_xt ∗tt)
  {
    if (¬Handler_Active) {
      warnx("SIGCHLD␣not␣handled");
      return;
    }
    else if (Handler_Active ≠ tt) {
      warnx("SIGCHLD␣not␣handled␣by␣this␣term_xt");
      return;
    }
    evsignal_del(tt→ev_signal);
    Handler_Active ← Λ;
  }
```

```
25. static void _term_reap(int fd, short e, void *arg)
  {
    term_xt *tt ← arg;
    int status;
    pid_t p;
    assert(e & EV_SIGNAL);
    assert(fd ≡ SIGCHLD);
    p ← waitpid(−1, &status, WNOHANG);
    if (p ≡ −1) err(1, "waitpid");
    else if (p ≡ 0) return;      /* But why are we here? */
    if (p ≠ tt→child_pid) {
      warn("wrong␣child␣pid␣%u?", p);
      return;
    }
    tt→exit_status ← status;
    tt→child_pid ← 0;
    if (¬IS_FLAG(tt→flags, TERM_IS_LIVE)) term_destroy(tt);      /* finish destruction */
    else if (tt→child_fd ≠ −1) {
      term_close_pty(tt);
      if (tt→billdoor) (tt→billdoor)(tt);
    }
    if (Handler_Active ≡ tt) term_remove_signal_handler(tt);
  }

26. bool term_has_pty(term_xt *tt)
  {
    return tt→child_fd ≠ −1;
  }
```

27. **bool** *term_attach_pty_exec*(**term_xt** ∗*tt*, **char** ∗*cmd*, **char** ∗∗*args*)

```
{
  char *no_args[] ← {Λ, Λ};
  struct passwd *pw;
  int m, s;
  pid_t p;
  assert(tt→child_pid ≡ 0);
  assert(tt→child_fd ≡ −1);
  if (¬args) args ← no_args;
  if (¬cmd) cmd ← args[0];
  if (¬cmd) {
    pw ← getpwuid(getuid());
    if (pw ∧ *pw→pw_shell) cmd ← pw→pw_shell;
    else {
      warn("getpwuid");
      cmd ← "/bin/sh";
    }
  }
  if (¬args[0]) args[0] ← cmd;
  if (openpty(&m, &s, Λ, Λ, Λ) ≡ −1) {
    warn("openpty");
    return false;
  }
  p ← fork();
  switch (p) {
  case −1: warn("fork");
    return false;
  case 0:     /* New process. */
    close(m);     /* Master side of pty */
    if (setsid() ≡ −1)     /* Become new session group leader. */
      err(1, "setsid");
    if (ioctl(s, TIOCSCTTY, Λ) < 0)     /* Controlling terminal. */
      err(1, "ioctl/TIOCSCTTY");
    dup2(s, 0);     /* Replace the child's stdio with the pty slave. */
    dup2(s, 1);
    dup2(s, 2);
    unsetenv("COLUMNS");
    unsetenv("LINES");
    unsetenv("TERMCAP");
    setenv("TERM", "turtle", 1);
    execvp(cmd, args);
    err(1, "execvp");
  default:     /* Turtle. */
    close(s);     /* Slave side of pty */
    tt→child_pid ← p;
    tt→child_fd ← m;
    _term_install_io_handler(tt);
    return true;
  }
}
```

28. **void** *term_close_pty*(**term_xt** *$*tt$*)

```
{
  int fd, rval;
  assert(tt→child_fd ≠ −1);
  _term_remove_io_handler(tt);
  fd ← tt→child_fd;
  tt→child_fd ← −1;
  errno ← 0;
  do rval ← close(fd); while (rval ≡ −1 ∧ errno ≡ EINTR);
  if (rval ≡ −1) warn("close");
}
```

29. **static void** *_term_install_io_handler*(**term_xt** *$*tt$*)

```
{
  tt→ev_io ← event_new(tt→loop, tt→child_fd, EV_READ | EV_PERSIST, _term_read, tt);
  event_add(tt→ev_io, Λ);
}
```

30. **static void** *_term_remove_io_handler*(**term_xt** *$*tt$*)

```
{
  event_del(tt→ev_io);
  if (tt→slow) evtimer_del(tt→slow_more);
}
```

31. Parsing Host Input. ECMA-48 describe how to extend ASCII with control sequences. ECMA-35 also describes similar types of code sequence for designating and invoking character sets. Collectively these have come to be known as escape sequence because they can, and sometimes do, always begin with the `ESC` code.

term_consume_bytes is a publicly-accessible wrapper around the parser that breaks the input into pieces small enough to fit into the terminal object's buffer.

```
⟨ Public API (term.o) 5 ⟩ +≡
  void term_consume_bytes(term_xt *, uint8_t *, size_t);
```

32. Parsing and interpreting those bytes uses these functions.

```
⟨ Function declarations (term.o) 9 ⟩ +≡
  static void _term_collect_byte(term_xt *, uint8_t, bool);
  static void _term_collect_param(term_xt *, uint8_t);
  static void _term_collect_prefix(term_xt *, uint8_t);
  static void _term_consume_bytes(term_xt *, size_t);
  static void _term_emit_byte(term_xt *, uint8_t);
  static void _term_emit_cp(term_xt *, ucp_xt);
  static void _term_exec_SGR_38(term_xt *, action_xt *, int, bool);
  static void _term_exec_c0(term_xt *, uint8_t);
  static void _term_exec_c1(term_xt *, uint8_t);
  static void _term_exec_csi(term_xt *, uint8_t);
  static void _term_exec_dcs(term_xt *);
  static void _term_exec_esc(term_xt *, uint8_t);
  static void _term_exec_osc(term_xt *);
  static void _term_reset_parse_state(term_xt *);
  static void _term_utfio_reset(term_xt *);
```

33.
```
void term_consume_bytes(term_xt *tt, uint8_t *buf, size_t len)
  {
    while (len > TERM_BUFSIZE) {
      term_consume_bytes(tt, buf, TERM_BUFSIZE);
      buf += TERM_BUFSIZE, len −= TERM_BUFSIZE;
    }
    memmove(tt→buf, buf, len);
    assert(len ≤ TERM_BUFSIZE);
    assert(¬IS_FLAG(tt→flags, TERM_PARSER_ACTIVE));
    SET_FLAG(tt→flags, TERM_PARSER_ACTIVE);
    _term_consume_bytes(tt, len);
    CLEAR_FLAG(tt→flags, TERM_PARSER_ACTIVE);
  }
```

34. ECMA-48 describes how to form control sequences but gives no guidance for dealing with input which does not conform to the standard. DEC made a popular line of terminals which did handle such errors which has since become the de-facto standard for doing so.

Paul Flo Williams has performed extensive research on real terminals to determine, along with documentation produced by DEC, exactly what this method is which is summarised in a nice diagram at https://vt100.net/emu/dec_ansi_parser. When I originally wrote the code to implement I remarked in these notes that it was about as large a spaghetti monster as I'm willing to write before making it table-driven. It has since grown a little.

The parser begins in the *flo_ground* state. If the buffer is empty while the parser is in another state then the location is remembered and the parser resumes from there when next called using the GCC assigned goto extension.

Readers who don't like nests of goto statements should jump ahead half a dozen pages to after the control sequences have been scanned.

```
static void _term_consume_bytes(term_xt *tt, size_t len)
{
  uint8_t bufscii;    /* The current byte masked with 0x7f. */
  void *next, *start;
  tt→at ← −1;    /* Overflows to 0 on first use. */
  if (tt→resume) {
    start ← tt→resume;
    tt→resume ← Λ;    /* unnecessary safety valve */
    goto *start;
  }
  ⟨ Ground state 36 ⟩
  ⟨ Ignored control strings 37 ⟩
  ⟨ Escape (ESC) sequences 40 ⟩
  ⟨ Control (CSI) sequences 42 ⟩
  ⟨ Device Control Strings (DCS) 46 ⟩
  ⟨ Operating System Command (OSC) control strings 47 ⟩
  abort();    /* UNREACHABLE */
}
```

35. The majority of states (except, notable, *flo_ground*) begin with *_term_flo_transition* which checks for reaching the end of the buffer and processes C0 and C1 control codes.

When the GROUND state is not *flo_ground* it is treated as the exit path from the current state, used by DCS and OSC strings. TODO: when exiting the DCS and OSC states (in situations that shouldn't terminate it normally) the control and DCS/OSC may apply in the wrong order.

```
#define DPTR(X) &&X    /* work around a CWEB deficiency */
#define _term_flo_next(RETURN) do
        {
          if (++tt→at ≥ len) {
            tt→resume ← DPTR(RETURN);
            return;
          }
        }
        while (0)
#define _term_flo_exit(GROUND, DESTINATION) do
        {
          if (DPTR(GROUND) ≡ DPTR(flo_ground)) goto DESTINATION;
          next ← DPTR(DESTINATION);
          goto GROUND;
        }
        while (0)
#define _term_flo_transition(GROUND, RETURN) do
        {
          _term_flo_next(RETURN);
          _lio_byte(tt, LIO_IN, &tt→buf[tt→at], 1);
          next ← DPTR(flo_ground);
          switch (ucp_stick(tt→buf[tt→at])) {
          case 1:    /* C0 controls */
            if (tt→buf[tt→at] ≡ 0x18 ∨ tt→buf[tt→at] ≡ 0x1a) goto GROUND;    /* CAN, SUB */
            else if (tt→buf[tt→at] ≡ 0x1b) _term_flo_exit(GROUND, flo_esc);    /* ESC */
            break;
          case 8:    /* C1 controls */    /* should go via GROUND first if it's not flo_ground */
            _term_exec_c1(tt, tt→buf[tt→at]);
            goto GROUND;
          case 9:    /* C1 controls */
            if (tt→buf[tt→at] ≡ 0x90) _term_flo_exit(GROUND, flo_dcs_entry);    /* DCS */
            else if (tt→buf[tt→at] ≡ 0x9b) _term_flo_exit(GROUND, flo_csi_entry);    /* CSI */
            else if (tt→buf[tt→at] ≡ 0x9c) goto GROUND;    /* ST */
            else if (tt→buf[tt→at] ≡ 0x9d) _term_flo_exit(GROUND, flo_osc_string);    /* OSC */
            else if (tt→buf[tt→at] ≡ 0x98 ∨ tt→buf[tt→at] ≥ 0x9e)    /* SOS, PM, APC */
              _term_flo_exit(GROUND, flo_apc_string);
            else {    /* 0x91–0x97, 0x99, 0x9a */
              _term_exec_c1(tt, tt→buf[tt→at]);
              goto GROUND;
            }
          }
          bufscii ← tt→buf[tt→at] & 0x7f;
        }
        while (0)
```

36. This is the state machine's entry point. This is the only state which doesn't immediately use *_term_flo_transition* because 8-bit C1 controls are indistinguishable from UTF-8 continuation bytes: if the parser is part-way through decoding a UTF-8 sequence then the next byte is emitted and the decoder will handle it.

```
⟨ Ground state 36 ⟩ ≡
flo_ground:
  if (IS_FLAG(tt→flags, TERM_UTF_HOST)) {
    _term_flo_next(flo_ground);
    if (tt→utf.status ≡ UTFIO_PROGRESS ∧ ucp_stick(tt→buf[tt→at]) > 1) {
      _lio_byte(tt, LIO_IN, &tt→buf[tt→at], 1);
      _term_emit_byte(tt, tt→buf[tt→at]);
      goto flo_ground;
    }
    tt→at −−;       /∗ prepare to re-scan the same byte ∗/
  }
  _term_flo_transition(flo_ground, flo_ground);
  if (ucp_stick(bufscii) ≤ 1) _term_exec_c0(tt, bufscii);
  else _term_emit_byte(tt, tt→buf[tt→at]);
  goto flo_ground;
```

This code is used in section 34.

37. Controls which instruct the terminal to expect a string but which are ignored all use this same routine. Amusingly ctlseqs mentions `APC` and `PM` but not `SOS` but the function in Xterm that does the ignoring is called `ParseSOS`.

```
⟨ Ignored control strings 37 ⟩ ≡
flo_apc_string: _term_flo_transition(flo_ground, flo_apc_string);
  goto flo_apc_string;
```

This code is used in section 34.

38. When the start of an escape or control sequence is detected the parser discards any previous work and prepares to collect data about a new sequence. The UTF-8 decoder is also reset which emits a replacement character in case a sequence is somehow started while UTF-8 is partly decoded.

The parser collects the arguments necessary to perform an action, hence these poorly thought out flag names.

```
#define TERM_ACTION_PARSE_ERROR 0
#define TERM_ACTION_FIRST_DIGIT 1
#define TERM_ACTION_EXEC_FLAG 2     /∗ unused ∗/
#define TERM_ACTION_FLAG_SET 3     /∗ unused ∗/
  static void _term_reset_parse_state(term_xt ∗tt)
  {
    memset((char ∗) &tt→parser + sizeof (tt→parser.data), 0, sizeof (tt→parser) − sizeof
        (tt→parser.data));
    SET_FLAG(tt→parser.flags, TERM_ACTION_FIRST_DIGIT);
    buf_clear(&tt→parser.data);
    if (buf_get_length(&tt→parser.data) > TERM_BUFSIZE) buf_realloc(&tt→parser.data, TERM_BUFSIZE);
    if (buf_get_length(&tt→parser.data)) ∗((char ∗) buf_ref(&tt→parser.data, 0)) ← 0;
    _term_utfio_reset(tt);
  }
  static void _term_utfio_reset(term_xt ∗tt)
  {
    if (tt→utf.status ≡ UTFIO_PROGRESS) _term_emit_cp(tt, UCP_REPLACEMENT);
    utfio_init(&tt→utf);
  }
```

39. No known control has more than two prefix bytes although ECMA-48 doesn't place any limits so the action object is limited to two prefix bytes and receiving a third is an error and the control will be parsed until it's complete but ignored.

```
static void _term_collect_prefix(term_xt *tt, uint8_t val)
{
  if (¬tt→parser.prefix[0]) tt→parser.prefix[0] ← val;
  else if (¬tt→parser.prefix[1]) tt→parser.prefix[1] ← val;
  else SET_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR);
}
```

40. The *flo_esc* state is entered after ESC is received in nearly every condition. In this state the terminal will act immediately on receipt of any C0 control (recall that any 8-bit C1 control has been handled already in *_term_flo_transition*) and treat bytes from sticks 4 and 5 as the corresponding 7-bit C1 control.

If the byte following ESC is not a C1 control then *flo_esc_intermediate* continues scanning the escape sequence without looking for a (7-bit) C1 control.

```
⟨ Escape (ESC) sequences 40 ⟩ ≡
flo_esc: _term_reset_parse_state(tt);
flo_esc_continue: _term_flo_transition(flo_ground, flo_esc_continue);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: _term_exec_c0(tt, tt→buf[tt→at]);
    goto flo_esc_continue;
  case 2: _term_collect_prefix(tt, bufscii);
    goto flo_esc_intermediate;
  case 3: case 4: goto flo_esc_dispatch;
  case 5:
    switch (bufscii) {
    case 0x50: goto flo_dcs_entry;      /* P, DCS */
    case 0x5b: goto flo_csi_entry;      /* [, CSI */
    case 0x5d: goto flo_osc_string;      /* ], OSC */
    case 0x5e:      /* ^ (caret), PM */
    case 0x5f:      /* _ (underscore), APC */
    case 0x58:      /* X, SOS */
      goto flo_apc_string;
    default: goto flo_esc_dispatch;
    }
  case 6: goto flo_esc_dispatch;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_esc_continue;      /* DEL */
    else goto flo_esc_dispatch;
  }
  goto flo_esc_continue;
flo_esc_intermediate: _term_flo_transition(flo_ground, flo_esc_intermediate);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: _term_exec_c0(tt, bufscii);
    goto flo_esc_intermediate;
  case 2: _term_collect_prefix(tt, bufscii);
    goto flo_esc_intermediate;
  case 4: case 5: case 3: case 6: goto flo_esc_dispatch;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_esc_intermediate;      /* DEL */
    else goto flo_esc_dispatch;
  }
  goto flo_esc_continue;
```

See also section 41.

This code is used in section 34.

41. TODO: don't call c1 here (should not be done in esc_intermediate) and roll the rest into the gotos above.

```
⟨ Escape (ESC) sequences 40 ⟩ +≡
flo_esc_dispatch:
  if (ucp_stick(bufscii) ≡ 4 ∨ ucp_stick(bufscii) ≡ 5) _term_exec_c1(tt, bufscii);
  else _term_exec_esc(tt, bufscii);
  goto flo_ground;
```

42. The CSI control, 0x9b or ESC [, is known as Control Sequence Introducer and begins a sequence similar in spirit to an escape sequence: an optional prefix of <=>?, zero or more numbers separated by ;, an optional flag and a final operator.

The prefix and final byte determine the operation so the flag byte is treated like a prefix.

```
⟨ Control (CSI) sequences 42 ⟩ ≡
flo_csi_entry: _term_reset_parse_state(tt);
flo_csi_entry_continue: _term_flo_transition(flo_ground, flo_csi_entry_continue);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: _term_exec_c0(tt, bufscii);
    goto flo_csi_entry_continue;
  case 2: _term_last_arg(&tt→parser);
    _term_collect_prefix(tt, bufscii);
    goto flo_csi_intermediate;
  case 3:
    if (bufscii ≡ 0x3a) goto flo_csi_param_param_start;      /* : */
    else if (bufscii ≥ 0x3c) _term_collect_prefix(tt, bufscii);      /* <=>? */
    else _term_collect_param(tt, bufscii);
    goto flo_csi_param;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_csi_entry_continue;      /* DEL */
  }
  goto flo_csi_dispatch;
flo_csi_ignore: _term_flo_transition(flo_ground, flo_csi_ignore);
  if (ucp_stick(bufscii) ≤ 1) _term_exec_c0(tt, bufscii);
  if (ucp_stick(bufscii) ≤ 3 ∨ bufscii ≡ 0x7f) goto flo_csi_ignore;
  goto flo_ground;
flo_csi_param: _term_flo_transition(flo_ground, flo_csi_param);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: _term_exec_c0(tt, bufscii);
    goto flo_csi_param;
  case 2: _term_last_arg(&tt→parser);
    _term_collect_prefix(tt, bufscii);
    goto flo_csi_intermediate;
  case 3:
    if (bufscii ≡ 0x3a) goto flo_csi_param_param_start;
    if (bufscii ≥ 0x3c) goto flo_csi_ignore;      /* :, <=>? */
    _term_collect_param(tt, bufscii);
    goto flo_csi_param;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_csi_param;      /* DEL */
  }
  _term_last_arg(&tt→parser);
  goto flo_csi_dispatch;
flo_csi_intermediate: _term_flo_transition(flo_ground, flo_csi_intermediate);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: _term_exec_c0(tt, bufscii);
    goto flo_csi_intermediate;
  case 2: _term_collect_prefix(tt, bufscii);
    goto flo_csi_intermediate;
  case 3: goto flo_csi_ignore;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_csi_intermediate;      /* DEL */
  }      /* else fall through */
```

See also sections 43 and 44.

This code is used in section 34.

43. The main `CSI` section is too big to fit on a single page.

```
⟨ Control (CSI) sequences 42 ⟩ +≡
flo_csi_dispatch: _term_exec_csi(tt, bufscii);
  goto flo_ground;
```

44. This is the last and ugliest wrinkle added to the state machine after it was first written. ECMA-48 allows control sequence parameters to include the `:` byte giving the example of representing a decimal point but little if anything used this feature until it was co-opted to support a wider range of colours in the `SGR` control.

```
⟨ Control (CSI) sequences 42 ⟩ +≡
flo_csi_param_param_start: _term_last_arg(&tt→parser);    /* not last; bad name */
  int n ← tt→parser.narg − 1;
  buf_push(&tt→parser.data, sizeof (tt→parser.arg[0]), &tt→parser.arg[n]);
    /* save the op parameter */
  tt→parser.arg[n] ← −buf_get_length(&tt→parser.data);
  _term_collect_byte(tt, 0, false);
  buf_pop(&tt→parser.data, 1, Λ);    /* push a string terminator in case there's no data to follow */
  goto flo_csi_param_param_continue;
flo_csi_param_param_continue: _term_flo_transition(flo_ground, flo_csi_param_param_continue);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: _term_exec_c0(tt, bufscii);
    goto flo_csi_param_param_continue;
  case 2: buf_push(&tt→parser.data, 1, Λ);    /* claim the extra 0 pushed in the last collect prefix */
    _term_collect_prefix(tt, bufscii);
    goto flo_csi_intermediate;
  case 3:
    if (bufscii ≥ 0x30 ∧ bufscii ≤ 0x3a) {
      _term_collect_byte(tt, bufscii, false);
      goto flo_csi_param_param_continue;
    }
    else if (bufscii ≡ 0x3b) {
      buf_push(&tt→parser.data, 1, Λ);    /* claim the extra 0 pushed in the last collect prefix */
      SET_FLAG(tt→parser.flags, TERM_ACTION_FIRST_DIGIT);
      goto flo_csi_param;
    }
    else    /* bufscii ≥ 0x3c */
      goto flo_csi_ignore;
  case 7:
    if (bufscii ≡ 0x7f) {
      buf_push(&tt→parser.data, 1, Λ);    /* claim the extra 0 pushed in the last collect prefix */
      goto flo_csi_intermediate;    /* DEL & 0xff */
    }
  }    /* else fall through */
  buf_push(&tt→parser.data, 1, Λ);    /* claim the extra 0 pushed in the last collect prefix */
  goto flo_csi_dispatch;
```

45. This routine increases the value of the current parameter or moves to collecting the next parameter. DEC state, and it's generally accepted, that 14 bits is a sufficient maximum size for each parameter however this routine parses into a larger value (**int**) and treats overflow of that (but *not* a number larger than 16384) as an error.

```
#define _term_arg(T, N, D)   (((T)→narg > (N) ∧ (T)→arg[N]) ? (T)→arg[N] : (D))
#define _term_next_arg(T)   do
          {
            if ((T)→narg ≡ TERM_ARG_SIZE) {
              warnx("too␣many␣arguments");
              SET_FLAG((T)→flags, TERM_ACTION_PARSE_ERROR);
            }
            else (T)→narg++;
          }
          while (0)
#define _term_last_arg(T)   do
          {
            if (IS_FLAG((T)→flags, TERM_ACTION_FIRST_DIGIT) ∧ (T)→narg)
              _term_next_arg(T);
            CLEAR_FLAG((T)→flags, TERM_ACTION_FIRST_DIGIT);
          }
          while (0)

  static void _term_collect_param(term_xt *tt, uint8_t val)
  {
    assert((val ≥ 0x30 ∧ val ≤ 0x39) ∨ val ≡ 0x3b);
    if (IS_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR)) return;
    if (IS_FLAG(tt→parser.flags, TERM_ACTION_FIRST_DIGIT)) {
      if (val ≠ 0x3b) {
        if (tt→parser.narg < TERM_ARG_SIZE) tt→parser.arg[tt→parser.narg] ← val & 0xf;
        CLEAR_FLAG(tt→parser.flags, TERM_ACTION_FIRST_DIGIT);
      }
      _term_next_arg(&tt→parser);
    }
    else {
      assert(tt→parser.narg ≥ 1 ∧ tt→parser.narg ≤ TERM_ARG_SIZE);
      int n ← tt→parser.narg − 1;
      if (val ≡ 0x3b) SET_FLAG(tt→parser.flags, TERM_ACTION_FIRST_DIGIT);
      else if (¬psnip_safe_mul(&tt→parser.arg[n], tt→parser.arg[n],
             10) ∨ ¬psnip_safe_add(&tt→parser.arg[n], tt→parser.arg[n], val & 0xf)) {
          /* tt→parser.arg[n] * 10 + (val & 0xf) */
        if (¬SET_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR)) warnx("argument␣overflow");
        SET_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR);
      }
    }
  }
```

46. Parsing `DCS` parameters is substantially similar to `CSI` parameters except that C0 controls are ignored and it's followed by a string. In the control string after the introducer sequence (DEC-STD-070 ss. 3.5.4.5), graphic characters with the high bit set (GR) are explicitly allowed and passed on to the consumer as-is. This doesn't include C1 controls.

```
⟨ Device Control Strings (DCS) 46 ⟩ ≡
flo_dcs_entry: _term_reset_parse_state(tt);
flo_dcs_entry_continue: _term_flo_transition(flo_ground, flo_dcs_entry_continue);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: goto flo_dcs_entry_continue;
  case 2: _term_collect_prefix(tt, bufscii);
    goto flo_dcs_entry_continue;
  case 3:
    if (bufscii ≡ 0x3a) goto flo_dcs_ignore;    /* : */
    else if (bufscii ≥ 0x3c) _term_collect_byte(tt, bufscii, false);    /* <=>? */
    else _term_collect_param(tt, bufscii);
    goto flo_dcs_param;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_dcs_entry_continue;
  }
  _term_collect_byte(tt, bufscii, false);
  goto flo_dcs_passthrough;
flo_dcs_ignore: _term_flo_transition(flo_ground, flo_dcs_ignore);
  goto flo_dcs_ignore;
flo_dcs_param: _term_flo_transition(flo_ground, flo_dcs_param);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: goto flo_dcs_param;
  case 2: _term_collect_prefix(tt, bufscii);
    goto flo_dcs_intermediate;
  case 3:
    if (bufscii ≡ 0x3a ∨ bufscii ≥ 0x3c) goto flo_dcs_ignore;    /* :, <=>? */
    _term_collect_param(tt, bufscii);
    goto flo_dcs_param;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_dcs_param;    /* DEL */
  }
  goto flo_dcs_passthrough;
flo_dcs_intermediate: _term_flo_transition(flo_ground, flo_dcs_intermediate);
  switch (ucp_stick(bufscii)) {
  case 0: case 1: goto flo_dcs_intermediate;
  case 2: _term_collect_prefix(tt, bufscii);
    goto flo_dcs_intermediate;
  case 3: goto flo_dcs_ignore;
  case 7:
    if (bufscii ≡ 0x7f) goto flo_dcs_intermediate;    /* DEL */
  }
  goto flo_dcs_passthrough;
flo_dcs_passthrough: _term_last_arg(&tt→parser);
  _term_flo_transition(flo_dcs_passthrough_end, flo_dcs_passthrough);
  if (bufscii ≠ 0x7f) _term_collect_byte(tt, bufscii, false);    /* DEL & 0xff */
  goto flo_dcs_passthrough;
flo_dcs_passthrough_end:
  if (tt→buf[tt→at] ≠ 0x18 ∧ tt→buf[tt→at] ≠ 0x1a) _term_exec_dcs(tt);    /* CAN, SUB */
  goto *next;
```

This code is used in section 34.

47. An `OSC` includes a control string like `DCS` which consists of arguments separated by `;` and it doesn't accept numeric parameters like a `CSI` sequence, so the list of arguments is re-used to contain offsets into the string buffer of the start of each string argument. See `OPT_BROKEN_ST` in xterm's `charproc.c` for more broken termination options that this terminal does not support.

```
⟨ Operating System Command (OSC) control strings 47 ⟩ ≡
flo_osc_string: _term_reset_parse_state(tt);
flo_osc_string_continue: _term_flo_transition(flo_osc_end, flo_osc_string_continue);
  if (bufscii ≡ 7) goto flo_osc_end;      /* BEL */
  if (ucp_stick(bufscii) > 1) _term_collect_byte(tt, bufscii, true);
  goto flo_osc_string_continue;
flo_osc_end:
  if (tt→buf[tt→at] ≠ 0x18 ∧ tt→buf[tt→at] ≠ 0x1a) _term_exec_osc(tt);      /* CAN, SUB */
  goto *next;
```

This code is used in section 34.

48. This routine expects that the high bit has already been masked off if necessary and ensures there is always a zero byte after the saved string (which is not included in the collected length). The `OSC` control which parses text arguments uses this routine to separate the string at `;` characters, which it replaces with a zero byte, and record the start of each argument in *tt→parser.arg*.

```
  static void _term_collect_byte(term_xt *tt, uint8_t val, bool arg)
  {
    uint8_t zeroed[2] ← {val, 0};
    if (arg ∧ val ≡ 0x3b) zeroed[0] ← 0;      /* ; */
    if (buf_push(&tt→parser.data, 2, zeroed)) buf_pop(&tt→parser.data, 1, Λ);
    else SET_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR);
    if (arg ∧ val ≡ 0x3b) {      /* ; */
      _term_next_arg(&tt→parser);
      if (tt→parser.narg < TERM_ARG_SIZE − 1)
        tt→parser.arg[tt→parser.narg] ← buf_get_used(&tt→parser.data);
    }
  }
```

49. Interpreting control sequences. If a byte doesn't represent part of a control sequence then it's emitted for display. The byte is always passed to the UTF-8 decoder if a sequence is in progress to detect an incomplete sequence.

If 7-bit character sets were going to be implemented it would be in here so a simple mapping routine is employed. A 7-bit code contains 94 or 96 characters not the full 128 so the byte value minus 32 is looked up in a table of 96 unicode code points. If that value is zero (or the set is really a 94-character set) then the byte value is used, hence the ASCII lookup table is a 94-character set of all zeros.

TODO: character sets because it's more or less complete and not hard.

```
static void _term_emit_byte(term_xt *tt, uint8_t val)
{
  int use, val7f;
  ucp_xt cp;
  assert(ucp_stick(val) > 1);
  if (tt→utf.status ≡ UTFIO_PROGRESS ∨ ((val & 0x80) ∧ (IS_FLAG(tt→flags, TERM_UTF_HOST)))) {
    switch (utfio_read(&tt→utf, val)) {
    case UTFIO_PROGRESS: return;
    default: fprintf(stderr, "utf8:␣discarding␣%x␣at␣%02x:␣%s\n", tt→utf.value, val,
          utfio_errormsg(&tt→utf));
      tt→utf.value ← UCP_REPLACEMENT;
    case UTFIO_COMPLETE: _term_emit_cp(tt, tt→utf.value);
      utfio_init(&tt→utf);
      return;
    }
  }
  else if (IS_FLAG(tt→flags, TERM_UTF_HOST)) {      /* never use GL/GR if unicode is expected */
    if (val) _term_emit_cp(tt, val);
  }
  else {
    use ← (val & 0x80) ? tt→use_gr : tt→use_gl;
    val7f ← val & 0x7f;
    assert(val7f ≥ 0x20);      /* nb. 0x20 and 0x7f are possible! */
    if ((val7f ≡ 0 ∨ val7f ≡ 0x7f) ∧ ¬tt→gr96[use]) cp ← 0;
    else if (¬(cp ← tt→grachar[use]→codepoint[val7f − 0x20])) cp ← val7f;
    if (cp) _term_emit_cp(tt, cp);
  }
}
```

50. When the terminal is ready to print a new code point the first and likely simplest problem is Automatic Margins or Auto Wrap Mode. Turtle follows the procedure outlined in DEC-STD-070 ss. D.6 which states that the cursor advances after a character is inserted unless it is already in the last column. If in the last column and Auto Wrap Mode is set then the Last Column Flag is set. If that flag is set when entering a character the active position will "advance to the first column of the next line PRIOR to entering that character", and reset the flag. If Auto Wrap Mode is not set then the new code point replaces the previous code point.

Other actions also reset (or save & restore) the Last Column Flag and are also listed (non exhaustively?) in DEC-STD-070 ss. D.6.

Insert/Replace Mode (IRM) is described at DEC-STD-070 p. 5-138. It does what it sounds like: pushes characters to the right of the cursor.

Even if *cp* looks like a control code, it's not. Control codes have all been handled and although unicode code points can have the same values as C1 (and C0) controls, the byte on the line that represented this code point were not.

```
static void _term_emit_cp(term_xt *tt, ucp_xt cp)
{
  int right;
  assert(¬ucp_isnonchar(cp));      /* Is 0xfffe or 0xffff in its plane? */
  assert(¬ucp_isnonbmp(cp));      /* Invalid region within the BMP? */
  assert(¬ucp_isnonrange(cp));      /* Outside 0–0x10ffff? */
  assert(¬ucp_issurrogate(cp));      /* Is a surrogate pair half? */
  if (IS_FLAG(tt→flags, TERM_AUTO_WRAP) ∧ IS_FLAG(tt→flags, TERM_LAST_COLUMN)) {
    assert(tt→panel→cursor.col ≡ tt→panel→right);
    tt→panel→cursor.col ← tt→panel→left;
    if (tt→panel→cursor.row < tt→panel→bottom) tt→panel→cursor.row++;
    else {
      panel_blit(tt→panel, tt→panel→top + 1, tt→panel→left, tt→panel→top, tt→panel→left,
          tt→panel→right − (tt→panel→left − 1), tt→panel→bottom − tt→panel→top);
      panel_clear_region(tt→panel, tt→panel→cursor.row, tt→panel→left,
          tt→panel→right − (tt→panel→left − 1), 1, 0);
    }
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    panel_set_here(tt→panel, cp);
  }
  else if (tt→panel→cursor.col ≡ tt→panel→right) {
    panel_set_here(tt→panel, cp);
    if (IS_FLAG(tt→flags, TERM_AUTO_WRAP)) SET_FLAG(tt→flags, TERM_LAST_COLUMN);
  }
  else {
    if (tt→panel→cursor.col > tt→panel→right) right ← tt→panel→width;
    else right ← tt→panel→right;
    if (IS_FLAG(tt→flags, TERM_INSERT_REPLACE))
      panel_blit(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col, tt→panel→cursor.row,
          tt→panel→cursor.col + 1, right − tt→panel→cursor.col, 1);
    panel_set_here(tt→panel, cp);
    tt→panel→cursor.col++;
  }
}
```

51. Parsing `ESC` sequences is mostly described in ECMA-35 with some details from DEC. In particular although the final byte determines what the sequence is, the choice of final byte is restricted by the first byte following the `ESC`. Recall that C0 controls, `0x7f` and bytes with the high bit set have all been handled by the parser state machine and will not reach here.

This section needs a lot of work because it's mostly concerned with unimplemented character sets and only the controls to switch in and out of UTF-8 mode and some others are correctly implemented.

A first byte that begins with 2 mostly concern character or code sets (eg. shifting) but see below.

A first byte beginning with 3 designates a private control function.

A first byte beginning with 4 or 5 is a C1 control.

If the first byte begins 6 or 7 (excluding `0x7f`) that's a final byte and the sequence is complete.

This is the table of `ESC` sequences with a 2-byte with corresponding labels from ECMA-35 table 3.b:

0F `ESC SP`: Announce code structure (15.2)
1F `ESC !`: Designate C0
2F `ESC "`: Designate C1
3F `ESC #`: Private control
4F `ESC $`: Multi-byte character set
5F `ESC %`: Designate other coding system
6F `ESC &`: Identify revised registration
7F `ESC '`: Reserved
8F `ESC (`: Designate G0-94
9F `ESC )`: Designate G1-94
10F `ESC *`: Designate G2-94
11F `ESC +`: Designate G3-94
12F `ESC ,`: Reserved
13F `ESC -`: Designate G1-96
14F `ESC .`: Designate G2-96
15F `ESC /`: Designate G3-96

Multi-byte character set (`ESC $`) does not refer to a UTF encoding but the legacy concept of multiple sets of 94 or 96 character Gn sets. Like Xterm this terminal specifies UTF-8 encoding with the standard method of designating a coding system incompatible with ECMA-35, `DOCS` (`ESC %`).

The final byte for each of 1F, 2F, 8-11F and 13-15F (and the related functions included in 4F) comes from a different namespace for each so despite appearances there is no possibility of the character sets overlapping (although of course they may be defined in terms of each other).

The meaning of `ESC SP F` and `ESC SP G`, toggling sending of 8-bit C1 controls, comes from ECMA-35 ss. 15.2.2 `ACS`.

ECMA-35 ss. 15.4 Designate Other Coding System (`DOCS`). The other coding system understood by this terminal is unicode in the form of UTF-8.

Another (or "an other") coding system is designated by `ESC %` followed by the system designation of one or two bytes. A `/` before the final byte indicates that the method of returning to an ECMA-35 coding system is *not* `ESC % @` (and that whatever does won't restore the state it was in). It's not clear where `G` to mean UTF-8 came from (because I haven't looked for it; registered with ISO-2375?).

TODO: The escape sequence parser collects intermediate bytes as prefix bytes of which there can be a maximum of two, making for escape sequences up to four bytes long (including `ESC`). Is this enough?

52. A C1 control only performs its action if there were no prefix bytes.

ECMA-6 ss. 9.2 ASCII IRV is designated by:

C0: ESC !

53. or ESC ! if absent. G0: ESC (B
BUT this cannot be sent per DOCS.

```
#define _p1(O, P) ((long)(O) | ((P) ≪ 8))
#define _p2(O, P, S) ((long)(O) | ((P) ≪ 8) | ((S) ≪ 16))
  static void _term_exec_esc(term_xt *tt, uint8_t final)
  {
    assert(final ≥ 0x20);
    assert(final < 0x40 ∨ final > 0x5f);
    assert(final < 0x7f);
    if (IS_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR) ∨ tt→parser.prefix[1]) return;
    if (tt→parser.prefix[0]) assert(ucp_stick(tt→parser.prefix[0]) ≡ 2);
    switch (tt→parser.prefix[0]) {
    case 0x20:    /* SP, ACS, ECMA-35 ss. 15.2.2 */
      if (¬tt→parser.prefix[1] ∧ (final ≡ 'F' ∨ final ≡ 'G')) {
        tt→parser.mode ← (final ≡ 'G');
        term_exec_SnC1T(tt, &tt→parser);
      }
      return;
    case 0x23:    /* #, Private use */
      switch (final) {
      case '3':    /* DECDHL, double-height (top). */
      case '4':    /* DECDHL, double-height (bottom). */
      case '5':    /* DECSWL, single-width. */
      case '6':    /* DECDWL, double-width. */
        return;
      case '8': term_exec_DECALN(tt, &tt→parser);
        return;
      default: return;
      }
    case 0x25: tt→parser.mode ← final;    /* % */
      term_exec_DOCS(tt, &tt→parser);
      return;
    case 0: break;    /* no prefix byte */
    default: return;
    }
    switch (final) {    /* TODO: These do not have prefix bytes (from ctlseqs): ESC 6 Back Index
          (DECBI), VT420 and up. ESC 7 Save Cursor (DECSC), VT100. ESC 8 Restore Cursor
          (DECRC), VT100. ESC 9 Forward Index (DECFI), VT420 and up. ESC = Application
          Keypad (DECKPAM). ESC > Normal Keypad (DECKPNM), VT100. ESC F Cursor to
          lower left corner of screen. This is enabled by the hpLowerleftBugCompat resource. ESC c
          Full Reset (RIS), VT100. ESC l Memory Lock (per HP terminals). Locks memory above
          the cursor. ESC m Memory Unlock (per HP terminals). ESC n Invoke the G2 Character
          Set as GL (LS2). ESC o Invoke the G3 Character Set as GL (LS3). ESC — Invoke the G3
          Character Set as GR (LS3R). ESC } Invoke the G2 Character Set as GR (LS2R). ESC ~
          Invoke the G1 Character Set as GR (LS1R), VT100. */
    default: return;
    }
  }
```

54. Turtle understands these C0 controls.

```
static void _term_exec_c0(term_xt *tt, uint8_t val)
{
  _term_utfio_reset(tt);
  switch (val) {
  case 0x00: break;     /* Do nothing. */
  case 0x07: term_exec_BEL(tt, &tt→parser);     /* ^G */
    break;
  case 0x08: term_exec_BS(tt, &tt→parser);     /* ^H */
    break;
  case 0x09: term_exec_HT(tt, &tt→parser);     /* ^I */
    break;
  case 0x0a: term_exec_LF(tt, &tt→parser);     /* ^J */
    break;
  case 0x0b: term_exec_VT(tt, &tt→parser);     /* ^K */
    break;
  case 0x0c: term_exec_FF(tt, &tt→parser);     /* ^L */
    break;
  case 0x0d: term_exec_CR(tt, &tt→parser);     /* ^M */
    break;
  }
}
```

55. Turtle understands these C1 controls in addition to the controls which affect the parser.

```
static void _term_exec_c1(term_xt *tt, uint8_t val)
{
  _term_utfio_reset(tt);
  switch (val) {
  case 'D': case 0x84: term_exec_IND(tt, &tt→parser);
    break;
  case 'E': case 0x85: term_exec_NEL(tt, &tt→parser);
    break;
  case 'H': case 0x88: term_exec_HTS(tt, &tt→parser);
    break;
  case 'M': case 0x8d: term_exec_RI(tt, &tt→parser);
    break;
  case '\\': case 0x9c: break;     /* ST; handled by the lexing state machine */
  }
}
```

56. These are the CSI sequence final and prefix bytes.

```
static void _term_exec_csi(term_xt *tt, uint8_t action)
{
  long arg;
  if (IS_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR)) return;
  switch (_p2(action, tt→parser.prefix[0], tt→parser.prefix[1])) {
  case '@': term_exec_ICH(tt, &tt→parser); break;
  case 'A': term_exec_CUU(tt, &tt→parser); break;
  case 'B': term_exec_CUD(tt, &tt→parser); break;
  case 'C': term_exec_CUF(tt, &tt→parser); break;
  case 'D': term_exec_CUB(tt, &tt→parser); break;
  case 'H': term_exec_CUP(tt, &tt→parser); break;
  case 'J': term_exec_ED(tt, &tt→parser); break;
  case _p1('J', '?'): term_exec_DECSED(tt, &tt→parser); break;
  case 'K': term_exec_EL(tt, &tt→parser); break;
  case _p1('K', '?'): term_exec_DECSEL(tt, &tt→parser); break;
  case 'L': term_exec_IL(tt, &tt→parser); break;
  case 'M': term_exec_DL(tt, &tt→parser); break;
  case 'P': term_exec_DCH(tt, &tt→parser); break;
  case 'S': term_exec_SU(tt, &tt→parser); break;
  case 'T': term_exec_SD(tt, &tt→parser); break;
  case 'X': term_exec_ECH(tt, &tt→parser); break;
  case 'c': term_exec_DA1(tt, &tt→parser); break;
  case _p1('c', '='): term_exec_DA3(tt, &tt→parser); break;
  case _p1('c', '>'): term_exec_DA2(tt, &tt→parser); break;
  case 'd': term_exec_VPA(tt, &tt→parser); break;
  case 'e': term_exec_VPR(tt, &tt→parser); break;
  case 'f': term_exec_HVP(tt, &tt→parser); break;
  case 'g': term_exec_TBC(tt, &tt→parser); break;
  case 'h':
    tt→parser.mode ← true;
    term_exec_SM(tt, &tt→parser);
    break;
  case _p1('h', '?'):
    tt→parser.mode ← true;
    term_exec_DECSET(tt, &tt→parser);
    break;
  case 'k': term_exec_VPB(tt, &tt→parser); break;
  case 'l':
    tt→parser.mode ← false;
    term_exec_SM(tt, &tt→parser);
    break;
  case _p1('l', '?'):
    tt→parser.mode ← false;
    term_exec_DECSET(tt, &tt→parser);
    break;
  case 'm': term_exec_SGR(tt, &tt→parser); break;
  case _p1('n', '?'): term_exec_DECXCPR(tt, &tt→parser); break;
  case 'q': break;    /* Blinkenlights! */
  case _p1('q', '>'):    /* report xterm name & version */
    break;
    ;    /* in theory prefix and suffix bytes are treated identically, so... */
  case _p1('q', '␣'):    /* set cursor style (vt520) */
  case _p1('q', '"'):    /* select character protection attribute (DECSCA) */
  case _p1('q', '#'):    /* pop video attributes from stack (XTPOPSGR), xterm */
    break;
  case 'r': term_exec_DECSTBM(tt, &tt→parser); break;
  case 's':
    if (IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN)) term_exec_DECSLRM(tt, &tt→parser);
    else return;      /* SCOSC, Save Cursor (ANSI) */
    break;
  case 't':
    arg ← _term_arg(&tt→parser, 0, 24);
    if (arg < 24) term_exec_XTWINOPS(tt, &tt→parser);
    else term_exec_DECSLPP(tt, &tt→parser);
    break;
  }
}
```

57. ECMA-48 doesn't define any device control strings. DEC does but I can't find an authoritative list. Xterm's ctlseqs lists the following:

`DCS Ps ; Ps | Pt ST`: User-Defined Keys (DECUDK).

`DCS $ q Pt ST`: Request Status String (DECRQSS), VT420 and up.

`DCS Ps $ t Pt ST`: Restore presentation status (DECRSPS), VT320 and up. This is the response to `DECRQSS`, not sent from the host.

`DCS + Q Pt ST`: Request resource values (XTGETXRES), xterm.

`DCS + p Pt ST`: Set Termcap/Terminfo Data (XTSETTCAP), xterm.

`DCS + q Pt ST`: Request Termcap/Terminfo String (XTGETTCAP), xterm.

From the parser: Stick 2 is optional prefix bytes. Stick 3 is optional parameters. Sticks 0-2 and 4-7 except essential C0s and DEL are collected except that the first byte must be sticks 4-7.

```
  static void _term_exec_dcs(term_xt *tt)
  {
    uint8_t action;
    if (IS_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR) ∨ ¬buf_get_length(&tt→parser.data))
      return;
    action ← *((char *) buf_ref(&tt→parser.data, 0));
    switch (_p2(action, tt→parser.prefix[0], tt→parser.prefix[1])) {
    case _p1('Q', '+'): term_exec_XTGETXRES(tt, &tt→parser); break;
    case _p1('q', '$'): term_exec_DECRQSS(tt, &tt→parser); break;
    case _p1('q', '+'): term_exec_XTGETTCAP(tt, &tt→parser); break;
    case '|': break;      /* DECUDK not implemented */
    case _p1('p', '+'): term_exec_XTSETTCAP(tt, &tt→parser); break;
    }
  }
```

58. *buf_ref*(&*tt*→*parser*.*data*, *tt*→*parser*.*arg*[*n*]) are Λ-terminated strings which should only consist of digits which describe the function to perform.

According to vt100.net vt520/525 terminals recognise OSC strings for `DECSIN` and `DECSWT`. Xterm lists one:

`OSC Ps ; Pt ST` (also terminated by `BEL`) named Set Text Parameters but notes that some controls also return information.

Xterm responds with the same terminator the host sent but note that every C1 will terminate the control string, not just ST.

DECSIN permits up to 12 characters: `OSC 2 L ; Pt ST`.

DECSWT permits up to 30 characters: `OSC 2 1 ; Pt ST`.

The function below was hacked together with ctlseqs open on the side. It probably interprets the controls listed there but might not stay in this form.

Nothing is implemented.

```
static void _term_exec_osc(term_xt *tt)
{
  size_t len;
  char *p;
  if (¬tt→parser.narg) {
    warnx("OSC␣has␣no␣arguments");
    return;
  }
  p ← buf_ref(&tt→parser.data, tt→parser.arg[0]);
  len ← strnlen(p, 4);
  if (¬len ∨ len ≡ 4) goto invalid_code;
  switch (p[0]) {
  case '0':
    if (len ≠ 1) goto invalid_code;    /* 1 arg; set icon name & window title */
    break;
  case '1':
    if (len ≡ 1) {    /* 1 */    /* 1 arg; set icon name */
    }
    else if (len ≡ 2) {    /* 10–19 */    /* 1 arg; change a colour */
    }
    else if (p[1] ≡ '1') {    /* 110–119 */    /* 0 arg; reset a colour */
    }
    else if (p[1] ≡ '0' ∧ p[2] ≡ '4') {    /* 104 */    /* 1 arg; reset chosen colour */
    }
    else if (p[1] ≡ '0' ∧ p[2] ≡ '5') {    /* 105 */    /* 1 arg; reset chosen special colour */
    }
    else if (p[1] ≡ '0' ∧ p[2] ≡ '6') {    /* 106 */
      /* 2 arg; enable/disable chosen special colour */
    }
    else goto invalid_code;
    break;
  case '2':
    if (len ≡ 3) goto invalid_code;
    else if (len ≡ 2 ∧ p[1] ≠ '2') goto invalid_code;
    else if (len ≡ 2) {    /* 22 */    /* 1 arg; change pointer chape */
    }
    else {    /* 2 */    /* 1 arg; set window title */
    }
    break;
  case '3':
    if (len ≠ 1) goto invalid_code;    /* 1 arg; set/clear X property */
    break;
  case '4':
    if (len ≡ 3) goto invalid_code;
    else if (len ≡ 2 ∧ p[1] ≠ '6') goto invalid_code;
    else if (len ≡ 2) {    /* 46 */    /* 1 arg; change log file */
    }
    else {    /* 4 */    /* x2 arg; change colour arg1 to spec arg2 or reply 'with a control
          sequence of the same form which can be used to set the corresponding color' if arg2 is '?' */
        /* if arg1 ¿ MAXCOLS (88 or 256), is equivalent to 5 with arg1 − MAXCOLS */
    }
    break;
  case '5':    /* or 50, 51, 52 */
    if (len ≡ 3) goto invalid_code;
    else if (len ≡ 2) {
      switch (p[1]) {
      case '0':    /* 50 */    /* 1 arg; set font */
        break;
      case '1':    /* 51 */    /* emacs breakage */
        break;
      case '2':    /* 52 */    /* 2 arg; selection */
        break;
      default: goto invalid_code;
      }
    }
    else {    /* 5 */    /* x2 arg; as 4 but change special colour 0–4 */
    }
    break;
  case '6':    /* or 60, 61 */
    if (len ≡ 3) goto invalid_code;
    else if (len ≡ 2) {
      switch (p[1]) {
      case '0':    /* 60 */    /* 0 arg; query allowed features */
        break;
      case '1':    /* 61 */    /* 0 arg; query disallowed features */
        break;
      default: goto invalid_code;
      }
    }
    else {    /* 6 */    /* equivalent to 106 */
    }
    break;    /* ctlseqs indicates that these three are “sun shelltool, cde dtterm”: */
  case 'I':    /* capital eye */
    if (len ≠ 1) goto invalid_code;    /* set icon to file */
    break;
  case 'l':    /* lower ell */
    if (len ≠ 1) goto invalid_code;    /* set window title */
    break;
  case 'L':    /* capital ell */
    if (len ≠ 1) goto invalid_code;    /* set icon label */
    break;
  default: invalid_code: warnx("invalid␣OSC␣code");
    return;
  }
}
```

59. Display Control Functions. This section defines the control functions described by DEC-STD-070 sections 3–6 and D, mostly in the order they're included in that document. Following the DEC controls come the controls defined later (mostly Xterm).

These are the controls that this terminal can perform.

⟨ Public API (term.o) 5 ⟩ +≡
```
void term_exec_BEL(term_xt *, action_xt *);
void term_exec_BS(term_xt *, action_xt *);
void term_exec_CPR(term_xt *, action_xt *);
void term_exec_CR(term_xt *, action_xt *);
void term_exec_CUB(term_xt *, action_xt *);
void term_exec_CUD(term_xt *, action_xt *);
void term_exec_CUF(term_xt *, action_xt *);
void term_exec_CUP(term_xt *, action_xt *);
void term_exec_CUU(term_xt *, action_xt *);
void term_exec_DA1(term_xt *, action_xt *);
void term_exec_DA2(term_xt *, action_xt *);
void term_exec_DA3(term_xt *, action_xt *);
void term_exec_DCH(term_xt *, action_xt *);
void term_exec_DECALN(term_xt *, action_xt *);
void term_exec_DECAWM(term_xt *, action_xt *);
void term_exec_DECBI(term_xt *, action_xt *);
void term_exec_DECCOLM(term_xt *, action_xt *);
void term_exec_DECFI(term_xt *, action_xt *);
void term_exec_DECLRMM(term_xt *, action_xt *);
void term_exec_DECOM(term_xt *, action_xt *);
void term_exec_DECRQSS(term_xt *, action_xt *);
void term_exec_DECSCPP(term_xt *, action_xt *);
void term_exec_DECSED(term_xt *, action_xt *);
void term_exec_DECSEL(term_xt *, action_xt *);
void term_exec_DECSET(term_xt *, action_xt *);
void term_exec_DECSLPP(term_xt *, action_xt *);
void term_exec_DECSLRM(term_xt *, action_xt *);
void term_exec_DECSTBM(term_xt *, action_xt *);
void term_exec_DECTCEM(term_xt *, action_xt *);
void term_exec_DECXCPR(term_xt *, action_xt *);
void term_exec_DL(term_xt *, action_xt *);
void term_exec_DOCS(term_xt *, action_xt *);
void term_exec_ECH(term_xt *, action_xt *);
void term_exec_ED(term_xt *, action_xt *);
void term_exec_EL(term_xt *, action_xt *);
void term_exec_FF(term_xt *, action_xt *);
void term_exec_HT(term_xt *, action_xt *);
void term_exec_HTS(term_xt *, action_xt *);
void term_exec_HVP(term_xt *, action_xt *);
void term_exec_ICH(term_xt *, action_xt *);
void term_exec_IL(term_xt *, action_xt *);
void term_exec_IND(term_xt *, action_xt *);
void term_exec_IRM(term_xt *, action_xt *);
void term_exec_LF(term_xt *, action_xt *);
void term_exec_LNM(term_xt *, action_xt *);
void term_exec_NEL(term_xt *, action_xt *);
void term_exec_RI(term_xt *, action_xt *);
void term_exec_SCS(term_xt *, action_xt *);
void term_exec_SD(term_xt *, action_xt *);
void term_exec_SGR(term_xt *, action_xt *);
void term_exec_SU(term_xt *, action_xt *);
void term_exec_SM(term_xt *, action_xt *);
void term_exec_SnC1T(term_xt *, action_xt *);
void term_exec_TBC(term_xt *, action_xt *);
void term_exec_VPA(term_xt *, action_xt *);
void term_exec_VPB(term_xt *, action_xt *);
void term_exec_VPR(term_xt *, action_xt *);
void term_exec_VT(term_xt *, action_xt *);
void term_exec_XTGETTCAP(term_xt *, action_xt *);
void term_exec_XTGETXRES(term_xt *, action_xt *);
void term_exec_XTSETTCAP(term_xt *, action_xt *);
void term_exec_XTWINOPS(term_xt *, action_xt *);
```

60. `S7C1T`. DEC-STD-070 p. 3-71, DEC-STD-070 ss. B-12. Select 7-bit C1 Transmission. `ESC SP F`.
`S8C1T`. DEC-STD-070 p. 3-72, DEC-STD-070 ss. B-12. Select 8-bit C1 Transmission. `ESC SP G`.
This escape sequence is also described in ECMA-35 ss. 15.2.2 as part of `ACS` Announce Code Structure.

```
void term_exec_SnC1T(term_xt *tt, action_xt *act)
{
  WAVE_FLAG(tt→flags, TERM_EIGHTBIT, act→mode);
}
```

61. `SI`, `LS0`. DEC-STD-070 p. 3-73. Shift In, Locking Shift Zero. C0 0x`0f`. nb. the DEC standard incorrectly says 0x`1f`.
`SO`, `LS1`. DEC-STD-070 p. 3-74. Shift Out, Locking Shift One. C0 0x`0e`. nb. the DEC standard incorrectly says 0x`1e`.
`LS2`. DEC-STD-070 p. 3-75. Locking Shift Two. `ESC n`.
`LS3`. DEC-STD-070 p. 3-76. Locking Shift Three. `ESC o`.
`LS1R`. DEC-STD-070 p. 3-77. Locking Shift One Right. `ESC ~`.
`LS2R`. DEC-STD-070 p. 3-78. Locking Shift Two Right. `ESC }`.
`LS3R`. DEC-STD-070 p. 3-79. Locking Shift Three Right. `ESC |`.
`SS2`. DEC-STD-070 p. 3-80. Single Shift Two. `ESC N`, C1 0x`8e`.
`SS3`. DEC-STD-070 p. 3-81. Single Shift Three. `ESC O`, C1 0x`8f`.
None of this is implemented.

62. `DOCS`. ECMA-35 ss. 15.4. Designate Other Coding System. `ESC %` .... It's not clear where the choice of `G` to identify UTF-8 came from.

```
void term_exec_DOCS(term_xt *tt, action_xt *act)
{
  assert(tt→parser.prefix[0] ≡ '%');
  if (tt→parser.prefix[1]) return;
  switch (act→mode) {
  case '@':    /* ECMA-35 */
    if (0)    /* not implemented */
      CLEAR_FLAG(tt→flags, TERM_UTF_HOST);
    break;
  case 'G':    /* UTF-8 */
    if (0)    /* not implemented */
      SET_FLAG(tt→flags, TERM_UTF_HOST);
    break;
  }
}
```

63. `DA/DA1`. ECMA-48 ss. 8.3.24, DEC-STD-070 p. 4-17, DEC-STD-070 ss. B.5. Device Attributes (Primary). `CSI Ps c`. Obsoletes `DECID`.

DEC-STD-070 ss. 4.5 makes clear that the first parameter returned in the response to this request indicates what the terminal is capable of not necessarily the conformance level it's currently operating at. The 4 in 64 indicates conformance level 4, which implies support for `DECSCL`, `DECSR`, 11, 14 & 17. Xterm advertises 1, 2, 6, 9, 15, 18, 21 and 22 from the list at DEC-STD-070 p. 4-19.

```
void term_exec_DA1(term_xt *tt, action_xt *act)
{
  if (act→narg > 1 ∨ _term_arg(act, 0, 0)) return;
  _term_init_reply(tt, act, TERM_ANSI_CSI, "?", 'c');
  act→arg[act→narg++] ← 64;    /* character cell display, level 4 */
  act→arg[act→narg++] ← 1;     /* 132 columns */
  act→arg[act→narg++] ← 21;    /* horizontal scrolling */
  _term_reply(tt, act, 0);
}
```

64. `DECSCL`. DEC-STD-070 p. 4-22. Select Conformance Level. `CSI Ps ; Ps " p`.
Turtle does not conform.

65. `DA2`. DEC-STD-070 p. 4-24. Device Attributes (Secondary). `CSI > Ps c`.
The VT terminals return a list of the product identity, firmware/software revision level and any optional extensions.

```
void term_exec_DA2(term_xt *tt, action_xt *act)
{
  _term_init_reply(tt, act, TERM_ANSI_CSI, ">", 'c');
  act→arg[act→narg++] ← 0;    /* product identification */
  act→arg[act→narg++] ← 0;    /* firmware/software revision */
  _term_reply(tt, act, 0);
}
```

66. `DA3`. DEC-STD-070 p. 4-26. Device Attributes (Tertiary). `CSI = c`.
Returned as a device control string (`DCS`). Xterm returns zero for the site code and serial number.

```
void term_exec_DA3(term_xt *tt, action_xt *act)
{
  _term_init_reply(tt, act, TERM_ANSI_DCS, "!", '|');
  buf_push(&act→data, 9, "00000000");
  _term_reply(tt, act, 0);
}
```

67. `DECID`.

68. `DECSR`.

69. `DECSTR`.

70. `DSR`.

71. `DECTCEM`. DEC-STD-070 p. 5-21. Text Cursor Enable Mode. `CSI ? 25 h`, `CSI ? 25 l`.
DEC suggest that when re-enabled the cursor remain off for an instant then begin its normal blinking cycle to ensure it blinks steadily. Terminfo `civis`, `cnorm`. Also `cvvis` (not implemented).

```
void term_exec_DECTCEM(term_xt *tt, action_xt *act)
{
  WAVE_FLAG(tt→flags, TERM_CURSOR_VISIBLE, act→mode);
}
```

72. `DECSTBM`. DEC-STD-070 p. 5-25, DEC-STD-070 ss. D.6. Set Top and Bottom Margins. `CSI Pt ; Pb r`. Terminfo `smgtb`.

```
void term_exec_DECSTBM(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  long top ← _term_arg(act, 0, 1);
  long bottom ← _term_arg(act, 1, tt→panel→height);
  if (top ≥ bottom ∨ bottom > tt→panel→height) return;
  tt→panel→top ← top;
  tt→panel→bottom ← bottom;
  if (IS_FLAG(tt→flags, TERM_ORIGIN_MOTION))
    tt→panel→cursor.row ← top, tt→panel→cursor.col ← tt→panel→left;
  else tt→panel→cursor.row ← tt→panel→cursor.col ← 1;
}
```

73. `DECSLRM`. DEC-STD-070 p. 5-27. Set Left and Right Margins. `CSI Pl ; Pr s`. Terminfo `smglr`.

```
void term_exec_DECSLRM(term_xt *tt, action_xt *act)
{
  if (¬IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN)) return;
  long left ← _term_arg(act, 0, 1);
  long right ← _term_arg(act, 1, tt→panel→width);
  if (left ≥ right ∨ right > tt→panel→width) return;
  tt→panel→left ← left;
  tt→panel→right ← right;
  if (IS_FLAG(tt→flags, TERM_ORIGIN_MOTION))
    tt→panel→cursor.row ← tt→panel→top, tt→panel→cursor.col ← tt→panel→left;
  else tt→panel→cursor.row ← tt→panel→cursor.col ← 1;
}
```

74. `DECLRMM`. DEC-STD-070 p. 5-29. Left/Right Margin Mode. `CSI ? 69 h`, `CSI ? 69 l`. This control doesn't have its own terminfo capability but is used as part of the `smglr` capability.

Although the DEC standard implies, and its example code does, reset the margins when `DECLRMM` is disabled, Xterm doesn't.

```
void term_exec_DECLRMM(term_xt *tt, action_xt *act)
{
  if (¬act→mode) {
    tt→panel→left ← 1;
    tt→panel→right ← tt→panel→width;
  }
  WAVE_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN, act→mode);
}
```

75. `DECOM`. DEC-STD-070 p. 5-31, DEC-STD-070 ss. D.6. Origin Mode. `CSI ? 6 h`, `CSI ? 6 l`. Whether absolute cursor positioning should refer to the corner of the display or accomodate the margins.

```
void term_exec_DECOM(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);      /* DEC-STD-070 ss. D.6 */
  WAVE_FLAG(tt→flags, TERM_ORIGIN_MOTION, act→mode);
}
```

76. `DECSCLM`. DEC-STD-070 p. 5-32. Scrolling Mode. `CSI ? 4 h`, `CSI ? 4 l`. Slows the display's scroll rate so that it's visible but not necessarily readable (not implemented) rather than scrolling at the maximum possible speed.

```
void term_exec_DECSCLM(term_xt *tt, action_xt *act)
{
  if (act→mode) printf("SLOW!\n");
}
```

77. `IND`. DEC-STD-070 p. 5-34 & DEC-STD-070 ss. D.6. Index. `ESC D`, C1 0x84. Although the `IND` control has been obsolete since the middle of the 20th century it's still implemented. DEC says that the VT100 and VT125 perform this identically to `LF` (ie. obey the New Line Mode flag).

```
#define _term_line_width(T, L)   ((T)→panel→width)     /* no double-wide characters */
  void term_exec_IND(term_xt *tt, action_xt *act)
  {
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);     /* DEC-STD-070 ss. D.6 */
    if (tt→panel→cursor.row < tt→panel→bottom) tt→panel→cursor.row++;
    else if (tt→panel→cursor.row ≡ tt→panel→bottom)
      panel_vscroll(tt→panel, 1, IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN));
    else if (tt→panel→cursor.row < tt→panel→height) tt→panel→cursor.row++;
    int16_t max ← _term_line_width(tt, tt→panel→cursor.row);
    if (tt→panel→cursor.col > max) tt→panel→cursor.col ← max;
  }
```

78. `RI`. ECMA-48 ss. 8.3.104, DEC-STD-070 p. 5-36 & DEC-STD-070 ss. D.6. Reverse Index. `ESC M`, C1 0x8d.

Trivia: Although it uses the acronym `RI` ECMA-48 names this control Reverse Line Feed and the Index control has been removed.

```
void term_exec_RI(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);     /* DEC-STD-070 ss. D.6 */
  if (tt→panel→cursor.row > tt→panel→top) tt→panel→cursor.row−−;
  else if (tt→panel→cursor.row ≡ tt→panel→top)
    panel_vscroll(tt→panel, −1, IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN));
  else if (tt→panel→cursor.row > 1) tt→panel→cursor.row−−;
  int16_t max ← _term_line_width(tt, tt→panel→cursor.row);
  if (tt→panel→cursor.col > max) tt→panel→cursor.col ← max;
}
```

79. `DECFI`. DEC-STD-070 p. 5-37. Forward Index. `ESC 9`. Move the cursor right, scrolling the display if necessary.

```
void term_exec_DECFI(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);     /* DEC-STD-070 ss. D.6 */
  if (tt→panel→cursor.col < tt→panel→right) tt→panel→cursor.col++;
  else if (tt→panel→cursor.col ≡ tt→panel→right)
    panel_hscroll(tt→panel, −1, IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN));
  else if (tt→panel→cursor.col < tt→panel→width) tt→panel→cursor.col++;
}
```

80. `DECBI`. DEC-STD-070 p. 5-39. Back Index. `ESC 6`. Move the cursor left, scrolling the display if necessary.

```
void term_exec_DECBI(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);      /* DEC-STD-070 ss. D.6 */
  if (tt→panel→cursor.col > tt→panel→left) tt→panel→cursor.col −−;
  else if (tt→panel→cursor.col ≡ tt→panel→left)
    panel_hscroll(tt→panel, 1, IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN));
  else if (tt→panel→cursor.col > 1) tt→panel→cursor.col −−;
}
```

81. `CUU`. ECMA-48 ss. 8.3.22, DEC-STD-070 p. 5-41 & DEC-STD-070 ss. D.6. Cursor Up. `CSI Pn A`. Terminfo `cuu`, `cuu1`.

```
void term_exec_CUU(term_xt *tt, action_xt *act)
{
  long n ← _term_arg(act, 0, 1);
  if (¬psnip_safe_sub(&n, tt→panel→cursor.row, n)) return;
  if (tt→panel→cursor.row ≥ tt→panel→top ∧ n < tt→panel→top) n ← tt→panel→top;
  else if (tt→panel→cursor.row < tt→panel→top ∧ n < 1) n ← 1;
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  tt→panel→cursor.row ← n;
  int16_t max ← _term_line_width(tt, tt→panel→cursor.row);
  if (tt→panel→cursor.col > max) tt→panel→cursor.col ← max;
}
```

82. `CUD`. ECMA-48 ss. 8.3.19, DEC-STD-070 p. 5-43 & DEC-STD-070 ss. D.6. Cursor Down. `CSI Pn B`. Terminfo `cud`, `cud1`.

```
void term_exec_CUD(term_xt *tt, action_xt *act)
{
  long n ← _term_arg(act, 0, 1);
  if (¬psnip_safe_add(&n, tt→panel→cursor.row, n)) return;
  if (tt→panel→cursor.row ≤ tt→panel→bottom ∧ n > tt→panel→bottom) n ← tt→panel→bottom;
  else if (tt→panel→cursor.row > tt→panel→bottom ∧ n < tt→panel→height) n ← tt→panel→height;
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  tt→panel→cursor.row ← n;
  int16_t max ← _term_line_width(tt, tt→panel→cursor.row);
  if (tt→panel→cursor.col > max) tt→panel→cursor.col ← max;
}
```

83. `CUF`. ECMA-48 ss. 8.3.20, DEC-STD-070 p. 5-45 & DEC-STD-070 ss. D.6. Cursor Right. `CSI Pn C`. Terminfo `cuf`, `cuf1`.

```
void term_exec_CUF(term_xt *tt, action_xt *act)
{
  long n ← _term_arg(act, 0, 1);
  if (¬psnip_safe_add(&n, tt→panel→cursor.col, n)) return;
  if (tt→panel→cursor.col ≤ tt→panel→right ∧ n > tt→panel→right) n ← tt→panel→right;
  else if (tt→panel→cursor.col > tt→panel→right ∧ n > tt→panel→width) n ← tt→panel→width;
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  tt→panel→cursor.col ← n;
}
```

84. `CUB`. ECMA-48 ss. 8.3.18, DEC-STD-070 p. 5-47 & DEC-STD-070 ss. D.6. Cursor Left. `CSI Pn D`. Terminfo `cub`, `cub1`.

```
void term_exec_CUB(term_xt *tt, action_xt *act)
{
  long n ← _term_arg(act, 0, 1);
  if (¬psnip_safe_sub(&n, tt→panel→cursor.col, n)) return;
  if (tt→panel→cursor.col ≥ tt→panel→left ∧ n < tt→panel→left) n ← tt→panel→left;
  else if (tt→panel→cursor.col < tt→panel→left ∧ n < 1) n ← 1;
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  tt→panel→cursor.col ← n;
}
```

85. `CUP`. ECMA-48 ss. 8.3.21, DEC-STD-070 p. 5-49 & DEC-STD-070 ss. D.6. Cursor Position. `CSI Pl ; Pc H`. Terminfo `cup`, `home`.

```
void term_exec_CUP(term_xt *tt, action_xt *act)
{
  long row ← _term_arg(act, 0, 1);
  long col ← _term_arg(act, 1, 1);
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  if (IS_FLAG(tt→flags, TERM_ORIGIN_MOTION)) {
    if (¬psnip_safe_add(&row, tt→panel→top − 1, row) ∨ ¬psnip_safe_add(&col, tt→panel→left − 1, col))
      return;
    if (row > tt→panel→bottom) row ← tt→panel→bottom;
    if (col > tt→panel→right) col ← tt→panel→right;
  }
  else {
    if (row > tt→panel→height) row ← tt→panel→height;
    if (col > tt→panel→width) col ← tt→panel→width;
  }
  tt→panel→cursor.row ← row;
  tt→panel→cursor.col ← col;
}
```

86. `HVP`. ECMA-48 ss. 8.3.63, DEC-STD-070 p. 5-51, DEC-STD-070 ss. D.6. Horizontal/Vertical Position. `CSI Pl ; Pc f`.

```
void term_exec_HVP(term_xt *tt, action_xt *act)
{
  term_exec_CUP(tt, act);      /* clears TERM_LAST_COLUMN */
}
```

87. `CPR`. ECMA-48 ss. 8.3.14, DEC-STD-070 p. 5-53. Cursor Position Report. `CSI 6 n`. Xterm assigns this to the `u7` terminfo capability explaining that it's "for the `tack` program".

```
void term_exec_CPR(term_xt *tt, action_xt *act)
{
  bool origin ← IS_FLAG(tt→flags, TERM_ORIGIN_MOTION);
  _term_init_reply(tt, act, TERM_ANSI_CSI, Λ, 'R');
  act→arg[act→narg++] ← tt→panel→cursor.row − (origin ? tt→panel→top − 1 : 0);
  act→arg[act→narg++] ← tt→panel→cursor.col − (origin ? tt→panel→left − 1 : 0);
  _term_reply(tt, act, 0);
}
```

88. `DECXCPR`. DEC-STD-070 p. 5-55. Extended Cursor Position Report. `CSI ? 6 n`.

```
void term_exec_DECXCPR(term_xt *tt, action_xt *act)
{
  bool origin ← IS_FLAG(tt→flags, TERM_ORIGIN_MOTION);
  _term_init_reply(tt, act, TERM_ANSI_CSI, "?", 'R');
  act→arg[act→narg++] ← tt→panel→cursor.row − (origin ? tt→panel→top − 1 : 0);
  act→arg[act→narg++] ← tt→panel→cursor.col − (origin ? tt→panel→left − 1 : 0);
  act→arg[act→narg++] ← 1;
  _term_reply(tt, act, 0);
}
```

89. `LNM`. DEC-STD-070 p. 5-57. New Line Mode. `CSI 20 h`, `CSI 20 l`. On a real terminal this control also causes the return key to transmit the `CR LF` sequence when set.

DEC notes that this mode should not be used in the set state.

```
void term_exec_LNM(term_xt *tt, action_xt *act)
{
  WAVE_FLAG(tt→flags, TERM_NEW_LINE_MODE, act→mode);
}
```

90. `CR`. ECMA-48 ss. 8.3.15, DEC-STD-070 p. 5-58 & DEC-STD-070 ss. D.6. Carriage Return. C0 0x0d (`^D`). Terminfo `cr`.

```
void term_exec_CR(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);      /* DEC-STD-070 ss. D.6 */
  if (tt→panel→cursor.col ≥ tt→panel→left) tt→panel→cursor.col ← tt→panel→left;
  else tt→panel→cursor.col ← 1;
}
```

91. `LF`. ECMA-48 ss. 8.3.74, DEC-STD-070 p. 5-59 & DEC-STD-070 ss. D.6. Line Feed. C0 0x0a (`^J`). Terminfo `ind`, but see the note there (viz. should it be?).

```
void term_exec_LF(term_xt *tt, action_xt *act)
{
  if (¬IS_FLAG(tt→flags, TERM_NEW_LINE_MODE)) term_exec_IND(tt, &tt→parser);
  else {
    term_exec_CR(tt, &tt→parser);
    if (tt→panel→cursor.row < tt→panel→bottom) tt→panel→cursor.row++;
    else if (tt→panel→cursor.row ≡ tt→panel→bottom)
      panel_vscroll(tt→panel, 1, IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN));
    else if (tt→panel→cursor.row < tt→panel→height) tt→panel→cursor.row++;
  }
}
```

92. `VT`. ECMA-48 ss. 8.3.161, DEC-STD-070 p. 5-61. Vertical Tab. C0 0x0b (`^K`). A printer would usually move the cursor down in a manner similar to horizontal tab stops.

```
void term_exec_VT(term_xt *tt, action_xt *act)
{
  term_exec_LF(tt, &tt→parser);
}
```

93. `FF`. ECMA-48 ss. 8.3.51, DEC-STD-070 p. 5-62. Form Feed. C0 0x0c (`^L`). A printer might move to the next page.

```
void term_exec_FF(term_xt *tt, action_xt *act)
{
  term_exec_LF(tt, &tt→parser);
}
```

94. `BS`. ECMA-48 ss. 8.3.5, DEC-STD-070 p. 5-63 & DEC-STD-070 ss. D.6. Back Space. C0 0x08 (`^H`). This is Back Space which moves the cursor backwards one space and implies nothing about what happens to the graphic characters under the cursor.

```
void term_exec_BS(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);      /* DEC-STD-070 ss. D.6 */
  if (tt→panel→cursor.col > tt→panel→left) tt→panel→cursor.col−−;
  else if (tt→panel→cursor.col < tt→panel→left ∧ tt→panel→cursor.col > 1) tt→panel→cursor.col−−;
}
```

95. `NEL`. ECMA-48 ss. 8.3.86, DEC-STD-070 p. 5-64 & DEC-STD-070 ss. D.6. Next Line. `ESC E`, C1 0x85.

```
void term_exec_NEL(term_xt *tt, action_xt *act)
{
  term_exec_CR(tt, &tt→parser);
  term_exec_IND(tt, &tt→parser);
}
```

96. `TBC`. ECMA-48 ss. 8.3.154, DEC-STD-070 p. 5-66. Tabulation Clear. `CSI Ps g`. Terminfo `tbc`. Also `it`. DEC (& xterm) implement 0 and 3; current position and all tabs. Other parameters refer to vertical tab stops (ECMA-48). Oddly there is no way to reset to the terminal's default tab stops.

```
void term_exec_TBC(term_xt *tt, action_xt *act)
{
  switch (_term_arg(act, 0, 0)) {
  case 0:      /* clear from current column */
    panel_remove_tabstop(tt→panel, tt→panel→cursor.col);
    break;
  case 3:      /* clear all */
    panel_remove_tabstop(tt→panel, 0);
    break;
  }
  return;
}
```

97. `HT`. ECMA-48 ss. 8.3.60, DEC-STD-070 p. 5-67. Horizontal Tab. C0 0x09 (`^I`). Terminfo `ht`.
TODO: What to do if there are no more tab stops?
TODO: The ANSI terminfo description uses `CSI I` which ctlseqs calls `CHT` (with Ps).

```
void term_exec_HT(term_xt *tt, action_xt *act)
{
  int to ← panel_find_tab(tt→panel, tt→panel→cursor.col);
  if (to ≤ tt→panel→right) tt→panel→cursor.col ← to;
  else if (tt→panel→cursor.col ≤ tt→panel→right) tt→panel→cursor.col ← tt→panel→right;
  else if (to > tt→panel→width) tt→panel→cursor.col ← to > tt→panel→width;
  else tt→panel→cursor.col ← to;
}
```

98. `HTS`. ECMA-48 ss. 8.3.62, DEC-STD-070 p. 5-69. Horizontal Tabulation Set. `ESC H`, C1 0x88.

```
void term_exec_HTS(term_xt *tt, action_xt *act)
{
  panel_insert_tabstop(tt→panel, tt→panel→cursor.col);
}
```

99. `DECCOLM`. DEC-STD-070 p. 5-71, DEC-STD-070 ss. D.6. Column Mode. `CSI ? 3 h`, `CSI ? 3 l`. There are no capabilities in terminfo corresponding to this feature but it's assumed to be present.

```
void term_exec_DECCOLM(term_xt *tt, action_xt *act)
{
  int w, h;
  struct winsize wsz ← {0};
  action_xt zero ← {0};
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  term_exec_CUP(tt, &zero);
  w ← act→mode ? 132 : 80;
  h ← 24;
  wsz.ws_row ← h;
  wsz.ws_col ← w;
  if (panel_resize(tt→panel, w, h, false)) {
    if (ioctl(tt→child_fd, TIOCSWINSZ, &wsz)) warn("ioctl/TIOCSWINSZ");
  }
  else term_exec_ED(tt, &zero);
}
```

100. TODO: Why is this here?

```
void _term_resize_tty(term_xt *tt)
{
  struct winsize wsz ← {0};
  wsz.ws_row ← tt→panel→height;
  wsz.ws_col ← tt→panel→width;
  if (tt→child_fd ≠ −1 ∧ ioctl(tt→child_fd, TIOCSWINSZ, &wsz)) warn("ioctl/TIOCSWINSZ");
}
```

101. `DECSCPP`. DEC-STD-070 p. 5-73. Set Columns Per Page. `CSI Pn $ |`.

```
void term_exec_DECSCPP(term_xt *tt, action_xt *act)
{
  long cols ← _term_arg(act, 0, 80);
  if (panel_resize(tt→panel, cols, tt→panel→height, true)) _term_resize_tty(tt);
}
```

102. `DECSLPP`. DEC-STD-070 p. 5-75. Set Lines Per Page. `CSI Pn t` where the (sole) argument is 24 or greater. Argument values below 24 have been co-opted to perform `XTWINOPS`

```
void term_exec_DECSLPP(term_xt *tt, action_xt *act)
{
  long rows ← _term_arg(act, 0, 24);
  if (panel_resize(tt→panel, tt→panel→width, rows, true)) _term_resize_tty(tt);
}
```

103. The rest of DEC-STD-070 ss. 5.4.6 and DEC-STD-070 ss. 5.4.7, pagination, is not implemented. `NP`–`PPB`.

104. DEC-STD-070 ss. 5.5, windowing, is not implemented. `DECHCCM`–`SD`.

105. DEC-STD-070 ss. 5.6, rendition, is not implemented yet but will be.
`DECSCNM`. Screen Mode.
`DECSWL`. Single-Width Line.
`DECDWL`. Double-Width Line.
`DECDHLT`. Double-Height Line Top.
`DECDHLB`. Double-Height Line Bottom.

106. `SGR`. DEC-STD-070 p. 5-103. Select Graphic Rendition. `CSI Ps ; ... ; Ps m`.

GRCM (ECMA-48) toggles between replacing and cumulative (doesn't affect case 0).

ECMA-48 and ctlseqs(*) also list:

* 1: bold * 2: faint or second colour * 3: italic * 4: underline x1 * 5: slow blink ¡150bpm 6: fast blink ¿=150bpm * 7: reverse * 8: invisible * 9: strikeout 10-20: font *21: underline x2 *22: Normal (neither bold nor faint), ECMA-48 3rd (ECMA-48 also mentions normal colour) *23: Not italicized, ECMA-48 3rd (ECMA-48 also not fraktur/gothic (20) but) *24: Not underlined, ECMA-48 3rd (x2 or x3) *25: Steady (not blinking), ECMA-48 3rd. 26: reserved by CCITT Rec. T.61 *27: Positive (not inverse), ECMA-48 3rd. *28: Visible, i.e., not hidden, ECMA-48 3rd, VT300. *29: Not crossed-out, ECMA-48 3rd. *30-49: colour inc. 38/48 reserved for future defined in ISO 8613-3/CCITT Rec. T.416; not in ctlseqs 50: undoes 26.

On 26 and 50: These are reserved by ECMA-48 for "proportional spacing" as specified by CCITT Recommendation T.61. CCITT, now ITU-T, has superceded T.61 with ....

E.3.2.2 p. 39 defines 26: "proportional spacing character pitch may be used" with note "at the recipient's option". Refers to a parameter value SHS to specify normal character pitch.

proportional support is indicated by a SGR with 26 only in handshake.

SHS on the next page with SVS (vertical spacing) too: character or lines per mm. Also SPD for direction, GSM Graphic Size Modification and SCO Character Orientation.

51: framed 52: encircled 53: overlined 54: undo 51,52 55: undo 53 56-59: reserved 60-65: ideogram under/right-lining and stress

ECMA-48 goes on to define SHS as (CSI Ps SP K) in 8.3.118 which defines the width of characters.

ctlseqs only: 90-97,100-107 duplicate 30-37,40-47.

ITU-T Rec. T.416 specifies `:` to separate parameters for SGRs 38 and 48 because SGR takes a list of attribute including 38 and 48. These parameters take parameters of their own and those are separated by `:`.

Also from T.416 ss. 8: Certain emphasis may be achieved by font selection (8.2).

Judging by T.416 SGR 10-19 designate a font and disregard "weight and posture". No way to invoke a non-designated font? Perhaps SGR 0?

T.416 deprecates 26/50.

T.412/ISO-8613-2 colour index (30-36 is 1-7, 37 is 0):

0 1,1,1 white 1 0,0,0 black 2 1,0,0 red 3 0,1,0 green 4 0,0,1 blue 5 1,1,0 yellow 6 1,0,1 mauve 7 0,1,1 cyan

Xterm OSC 4 (set-colour-index-to) matches the order in SGR.

DEC-STD-070 ss. 8.6.2.1 in discussing ReGIS lists (this matches the QL):

0 black 1 blue 2 red 3 magenta 4 green 5 cyan 6 yellow 7 white

Sixel (DEC-STD-070 ss. 9) has colour introducer alt to Xterm's OSC 4 (but also invokes that colour). Conforming software will use colours 0–255.

9.12 intro ends with an ordered list of colours:

black red green yellow blue magenta cyan white

Apart from swapping blue and yellow this is the same as the T.412 and SGR order.

CGA:

black blue green cyan red magenta yellow (brown) white

```
#define _term_subarg_offset(A, I)  (−(A)→arg[I])
#define _term_subarg_base(A, I)  (_term_subarg_offset((A), (I)) − sizeof ((A)→arg[I]))
#define _term_subarg_param(A, I)  (∗(int ∗) buf_ref(&(A)→data, _term_subarg_base((A), (I))))
#define _term_subarg_string(A, I)  ((uint8_t ∗) buf_ref(&(A)→data, _term_subarg_offset((A), (I))))

  void term_exec_SGR(term_xt ∗tt, action_xt ∗act)
  {
    if (¬act→narg) {
      act→arg[0] ← 0;
      act→narg ← 1;
    }
    for (int i ← 0; i < act→narg; i++) {
      if (act→arg[i] < 0)
        switch (_term_subarg_param(act, i)) {
        case 38: _term_exec_SGR_38(tt, act, i, false);
      break;
    case 48: _term_exec_SGR_38(tt, act, i, true);
      break;
    }
  else
    switch (act→arg[i]) {
    case 0: panel_reset_cell_flags(tt→panel);
      panel_reset_pen_colour(tt→panel, true, true);
      break;
    case 1:     /* bold */
      break;
    case 4: panel_begin_cell_underline_one(tt→panel);
      break;
    case 5: panel_begin_cell_blink_slow(tt→panel);
      break;
    case 6: panel_begin_cell_blink_fast(tt→panel);
      break;
    case 7: panel_begin_cell_inverse(tt→panel);
      break;
    case 8: panel_begin_cell_hidden(tt→panel);
      break;
    case 9: panel_begin_cell_strikeout(tt→panel);
      break;
    case 21: panel_begin_cell_underline_two(tt→panel);
      break;
    case 24: panel_end_cell_underline(tt→panel);
      break;
    case 25: panel_end_cell_blink(tt→panel);
      break;
    case 27: panel_end_cell_inverse(tt→panel);
      break;
    case 28: panel_end_cell_hidden(tt→panel);
      break;
    case 29: panel_end_cell_strikeout(tt→panel);
      break;
    case 30: case 31: case 32: case 33: case 34: case 35: case 36: case 37:
      panel_set_pen_colour_index(tt→panel, false, act→arg[i] − 30);
      break;
    case 39: panel_reset_pen_colour(tt→panel, true, false);
      break;
    case 40: case 41: case 42: case 43: case 44: case 45: case 46: case 47:
      panel_set_pen_colour_index(tt→panel, true, act→arg[i] − 40);
      break;
    case 49: panel_reset_pen_colour(tt→panel, false, true);
      break;
    case 53: panel_begin_cell_overline(tt→panel);
      break;
    case 55: panel_end_cell_overline(tt→panel);
      break;
    case 90: case 91: case 92: case 93: case 94: case 95: case 96: case 97:
      panel_set_pen_colour_index(tt→panel, false, act→arg[i] − 82);
      break;
    case 100: case 101: case 102: case 103: case 104: case 105: case 106: case 107:
      panel_set_pen_colour_index(tt→panel, true, act→arg[i] − 92);
      break;
    }
    }
  }
```

107. **static void** *_term_exec_SGR_38*(**term_xt** $*tt$, **action_xt** $*act$, **int** *arg*, **bool** *isbg*)

```
  {
    int n ← 0, p[8] ← {0};
    uint8_t *buf ← _term_subarg_string(act, arg);
    while (*buf) {
      assert(*buf ≥ 0x30 ∧ *buf ≤ 0x3a);
      if (*buf ≡ 0x3a) {
        if (++n ≥ 8) {
          warnx("sgr38␣excess");
          return;
        }
        buf ++;
        continue;
      }
      if (¬psnip_safe_mul(&p[n], p[n], 10) ∨ ¬psnip_safe_add(&p[n], p[n], *buf & 0xf)) {
          /* p[n] * 10 + (*buf & 0xf) */
        warnx("sgr38␣overflow");
        return;
      }
      buf ++;
    }
    if (¬n) return;
    switch (p[0]) {      /* CSID, ignored, defines scale of RGBCMYK; no space for alpha channel */
    case 0:      /* implementation-defined fg */
      case 1:      /* transparent, no args */
      case 3:      /* CMY: CSID,C,M,Y,-,tolerance */
      case 4:      /* CMYK: CSID,C,M,Y,K */
      break;
    case 2:      /* RGB: CSID,R,G,B,-,tolerance */
      if (p[1] ∨ p[2] > 255 ∨ p[3] > 255 ∨ p[4] > 255) {
        warnx("sgr38␣invalid");
        return;
      }
      panel_set_pen_colour_rgba(tt→panel, isbg, p[2], p[3], p[4], 255);
      break;
    case 5:      /* indexed */
      if (n > 2 ∨ p[1] > 255) {
        warnx("sgr38␣indexed␣invalid");
        return;
      }
      panel_set_pen_colour_index(tt→panel, isbg, p[1]);
      break;
    }
  }
```

108. ...
`DECCTR`. Color Table Report.
`DECSTGLT`. Select Text/Graphics Look-Up Table.

109. `BEL`. ECMA-48 ss. 8.3.3, DEC-STD-070 p. 5-114. Bell. C0 0x07 (`^G`). Terminfo `bel`.

```
void term_exec_BEL(term_xt *tt, action_xt *act)
{
  if (tt→bell) (tt→bell)(tt);
}
```

110. DEC-STD-070 ss. 5.8, graphic character sets, is only minimally implemented in *_term_emit_cp*. `SCS`, `DECAUPSS`, `DECRQUPSS`, `DECNRCM`,

111. DEC-STD-070 ss. 5.9. Mopping up remaining C0 & C1 controls.
ENQ XON XOFF DECID

112. `SUB`.

113. DEC-STD-070 ss. 5.10. Much of the functionality described here is obsolete. Most of the rest isn't implemented.

```
void term_exec_SM(term_xt *tt, action_xt *act)
{
  assert(¬IS_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR));
  for (int i ← 0; i < act→narg; i++) {
    switch (act→arg[i]) {
    case 4: term_exec_IRM(tt, act);
      break;
    case 20: term_exec_LNM(tt, act);
      break;     /* vttest (also) toggles these when it starts: */
    case 2:     /* KAM */
      case 12:     /* SRM */
      break;
    }
  }
}
```

114. `DECSET` is another name for the extension to `SM`/`RM` described in DEC-STD-070 ss. 5.10.1.2.

```
void term_exec_DECSET(term_xt *tt, action_xt *act)
{
  assert(¬IS_FLAG(tt→parser.flags, TERM_ACTION_PARSE_ERROR));
  for (int i ← 0; i < act→narg; i++) {
    switch (act→arg[i]) {
    case 3: term_exec_DECCOLM(tt, act); break;
    case 4: term_exec_DECSCLM(tt, act); break;
    case 6: term_exec_DECOM(tt, act); break;
    case 7: term_exec_DECAWM(tt, act); break;
    case 25: term_exec_DECTCEM(tt, act); break;
    case 69: term_exec_DECLRMM(tt, act); break;
        /* vttest (also) toggles these when it starts: */
    case 1:     /* DECCKM — Application (set) or normal cursor keys */
        /* See DEC STD 070 . 6.68, p. 6-60 */
      case 5:     /* DECSCNM — Normal or reverse (se) video */
      case 8:     /* DECARM — Auto-Repeat Keys */
      case 40:     /* xterm — Allow 80 to 132 mode */
      case 45:     /* xterm — Reverse-wraparound mode */
      break;     /* Ignored */
    }
  }
}
```

115. `IRM`. DEC-STD-070 p. 5-138. Insert/Replacement Mode. `CSI 4 h`, `CSI 4 l`. Terminfo `mir`, `rmir`, `smir` but `terminfo(5)` pairs them with a 700 word caveat.

Xterm uses the rmir cap in xterm-basic to reset (replace) and smir to set (insert) this mode.

The state of this flag affects character output in *_term_emit_cp*.

```
void term_exec_IRM(term_xt *tt, action_xt *act)
{
  WAVE_FLAG(tt→flags, TERM_INSERT_REPLACE, act→mode);
}
```

116. `ICH`. ECMA-48 ss. 8.3.64, DEC-STD-070 p. 5-142, DEC-STD-070 ss. D.6. Insert Character. `CSI Pn @`. Terminfo `ich`, `ich1`. DEC state in DEC-STD-070 ss. D.6 that this control clears the `TERM_LAST_COLUMN` flag although the cursor does not move.

```
void term_exec_ICH(term_xt *tt, action_xt *act)
{
  if (tt→panel→cursor.col < tt→panel→left ∨ tt→panel→cursor.col > tt→panel→right) return;
  int max ← tt→panel→right − tt→panel→cursor.col + 1;
  long n ← _term_arg(act, 0, 1);
  if (n > max) n ← max;
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  panel_blit(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col, tt→panel→cursor.row,
      tt→panel→cursor.col + n, max − n, 1);
  panel_clear_region(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col, n, 1, 0);
}
```

117. `DCH`, DEC-STD-070 ss. D.6. ECMA-48 ss. 8.3.26, DEC-STD-070 p. 5-144. Delete Character. `CSI Pn P`. Terminfo `dch`, `dch1`. DEC state in DEC-STD-070 ss. D.6 that this control clears the `TERM_LAST_COLUMN` flag although the cursor does not move.

```
void term_exec_DCH(term_xt *tt, action_xt *act)
{
  if (tt→panel→cursor.col < tt→panel→left ∨ tt→panel→cursor.col > tt→panel→right) return;
  int max ← tt→panel→right − tt→panel→cursor.col + 1;
  long n ← _term_arg(act, 0, 1);
  if (n > max) n ← max;
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  panel_blit(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col + n, tt→panel→cursor.row,
      tt→panel→cursor.col, max − n, 1);
  panel_clear_region(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col + max − n, n, 1, 0);
}
```

118. `IL`. ECMA-48 ss. 8.3.67, DEC-STD-070 p. 5-146, DEC-STD-070 ss. D.6. Insert Line. `CSI Pn L`. Terminfo `il`, `il1`. No need to test `TERM_LEFT_RIGHT_MARGIN` — if it's not set then left and right will be the panel's extremes, which the cursor cannot go beyond.

```
void term_exec_IL(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  int max ← tt→panel→bottom − tt→panel→cursor.row + 1;
  long n ← _term_arg(act, 0, 1);
  if (n > max) n ← max;
  panel_insert_line_here(tt→panel, n, true);
}
```

119. `DL`, DEC-STD-070 ss. D.6. ECMA-48 ss. 8.3.32, DEC-STD-070 p. 5-148. Delete Line. `CSI Pn M`. Terminfo `dl`, `dl1`.

```
void term_exec_DL(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  int max ← tt→panel→bottom − tt→panel→cursor.row + 1;
  long n ← _term_arg(act, 0, 1);
  if (n > max) n ← max;
  panel_delete_line_here(tt→panel, n, true);
}
```

120. `DECIC`. DEC-STD-070 p. 5-150. Insert Column. `CSI Pn ' }`.
`DECDC`. DEC-STD-070 p. 5-151. Delete Column. `CSI Pn ' ~`.
Curiously these are not called out for reseting last column mode by DEC-STD-070 ss. D.6.

121. `ECH`. ECMA-48 ss. 8.3.38, DEC-STD-070 p. 5-152, DEC-STD-070 ss. D.6. Erase Character. `CSI Pn X`. Terminfo `ech`. This control is not affected by the margins or insert/replace mode.

```
void term_exec_ECH(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  int max ← tt→panel→width − tt→panel→cursor.col + 1;
  long n ← _term_arg(act, 0, 1);
  if (n > max) n ← max;
  panel_clear_region(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col, n, 1, 0);
}
```

122. `EL`. ECMA-48 ss. 8.3.41, DEC-STD-070 p. 5-154, DEC-STD-070 ss. D.6. Erase In Line. `CSI Pn K`. Terminfo `el`, `el1`. This control is not affected by the margins.

```
void term_exec_EL(term_xt *tt, action_xt *act)
{
  switch (_term_arg(act, 0, 0)) {
  case 0:     /* cursor to end */
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    panel_clear_region(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col,
        tt→panel→width − tt→panel→cursor.col + 1, 1, 0);
    break;
  case 1:     /* cursor to beginning */
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    panel_clear_region(tt→panel, tt→panel→cursor.row, 1, tt→panel→cursor.col, 1, 0);
    break;
  case 2:     /* whole line */
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    panel_clear_region(tt→panel, tt→panel→cursor.row, 1, tt→panel→width, 1, 0);
    break;
  }
}
```

123. `ED`. ECMA-48 ss. 8.3.39, DEC-STD-070 p. 5-156, DEC-STD-070 ss. D.6. Erase in Display. `CSI Pn J`. Terminfo `clear`, `ed`. This control is not affected by the margins or origin mode.

```
void term_exec_ED(term_xt *tt, action_xt *act)
{
  switch (_term_arg(act, 0, 0)) {
  case 0:    /* cursor to end */
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    panel_clear_region(tt→panel, tt→panel→cursor.row, tt→panel→cursor.col,
        tt→panel→width − tt→panel→cursor.col + 1, 1, 0);
    if (tt→panel→cursor.row < tt→panel→height) panel_clear_region(tt→panel,
          tt→panel→cursor.row + 1, 1, tt→panel→width, tt→panel→height − tt→panel→cursor.row, 0);
    break;
  case 1:    /* cursor to beginning */
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    panel_clear_region(tt→panel, tt→panel→cursor.row, 1, tt→panel→cursor.col, 1, 0);
    panel_clear_region(tt→panel, 1, 1, tt→panel→width, tt→panel→cursor.row − 1, 0);
    break;
  case 2:    /* whole display */
    CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    panel_clear(tt→panel, 0);
    break;
  case 3: CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
    break;    /* Erase saved lines (xterm) */
  }
}
```

124. `DECSEL`. DEC-STD-070 p. 5-159. Selective Erase In Line. `CSI ? Pn K`.
`DECSED`. DEC-STD-070 p. 5-162. Selective Erase In Display. `CSI ? Pn J`.
Without selectively erasable characters this is identical to `EL` and `ED` except that they're ignored in level 1 operation, and appear to not reset the rendition.

```
void term_exec_DECSEL(term_xt *tt, action_xt *act)
{
  term_exec_EL(tt, &tt→parser);
}
void term_exec_DECSED(term_xt *tt, action_xt *act)
{
  term_exec_ED(tt, &tt→parser);
}
```

125. DEC-STD-070 ss. 5.12, on line performance improvements, is not implemented. `DECCRA`–`DECSACE`. Possibly a useful interface to panel blitting.

126. DEC-STD-070 ss. 5.13, saving and restoring state, is not implemented yet but will be.
`DECSC`. Save Cursor.
`DECRC`. Restore Cursor.
`DECRQM`. Request Mode.
`DECRPM`. Report Mode.

127. `DECRQSS`. Request Selection or Setting. `DCS $ q Pt ST`.

This has been implemented hackily based on the Xterm implementation because I couldn't find the definition in DEC-STD-070.

Response is `DECRSPS`: `DCS Ps $ t Pt ST`. Ps is 0 (valid) or 1 (invalid) request. Pt depends on the request, but is the CSI sequence without the CSI leader which will re-apply the state queried in the `DECRQSS`' Pt:

m SGR " p DECSCL SP q DECSCUSR " q DECSCA r DECSTBM s DECSLRM t DECSLPP $ — DECSCPP $ } DECSASD $ ␣ DECSSDT * — DECSNLS

This terminal only recognises `DECSTBM` and `DECSLRM` (it's not clear if the SLRM setting should include `DECLRMM`).

"The terminal does not send a data string (D...D) [Pt] to the host when the terminal receives an invalid request"

Xterm says something completely different about Ps than the programming manual does. It responds with a 1 which the manual says should mean the request was invalid.

```
void term_exec_DECRQSS(term_xt *tt, action_xt *act)
{
  char p[2];
  if (tt→parser.narg) return;
  memmove(p, buf_ref(&tt→parser.data, 1), 2);
  if (buf_get_length(&tt→parser.data) > 3) {
  missing: _term_init_reply(tt, act, TERM_ANSI_DCS, "1$", 't');
    buf_clear(&tt→parser.data);      /* no return */
  }
  else {
    _term_init_reply(tt, act, TERM_ANSI_DCS, "0$", 't');
    switch (_p1(p[0], p[1])) {
    default: case 'm':    /* SGR */
      case _p1('"', 'p'):    /* DECSCL */
      case _p1('.', 'q'):    /* DECSCUSR */
      case _p1('"', 'q'):    /* DECSCA */
      case 't':    /* DECSLPP */
      case _p1('$', '|'):    /* DECSCPP */
      case _p1('$', '}'):    /* DECSASD */
      case _p1('$', '~'):    /* DECSSDT */
      case _p1('*', '|'):    /* DECSNLS */
      goto missing;
    case 'r':    /* DECSTBM */
      printf("ignore␣DECSTBM\n");
      break;
    case 's':    /* DECSLRM */
      printf("ignore␣DECSLRM\n");
      break;
    }
  }
  _term_reply(tt, act, 0);
}
```

128. `DECRQPSR`. Request Presentation State Report.
`DECPSR`. Presentation State Report.
`DECCIR`. Cursor Information Report.
`DECTABSR`. Tabulation Stop Report.
`DECRSPS`. Restore Presentation State.
`DECRQTSR`. Request Terminal State Report.
`DECTSR`. Terminal State Report.
`DECRSTS`. Restore Terminal State.

129. `DECAWM`. DEC-STD-070 ss. D.6. Auto Wrap Mode. `CSI ? 7 h`, `CSI ? 7 l`. Terminfo `rmam`, `smam`.

```
void term_exec_DECAWM(term_xt *tt, action_xt *act)
{
  CLEAR_FLAG(tt→flags, TERM_LAST_COLUMN);
  WAVE_FLAG(tt→flags, TERM_AUTO_WRAP, act→mode);
}
```

130. `DECALN`. DEC-STD-070 p. D-19. Screen Alignment. `ESC # 8`.

Although it's not mentioned in DEC-STD-070 ss. D.8 (which says `DECALN` is not for general use), xterm fills the display with `E` characters which is presumably what real terminals did. DEC include an example for how to implement the algorithm which merely empties each cell (among others, in short it clears the screen).

```
void term_exec_DECALN(term_xt *tt, action_xt *act)
{
  act→narg ← 0;
  term_exec_CUP(tt, act);
  term_exec_ED(tt, act);      /* TODO: AWM? */
  for (int col ← 1; col ≤ tt→panel→width; col++)
    for (int row ← 1; row ≤ tt→panel→height; row++) panel_set_at(tt→panel, row, col, 'E');
  tt→slow_budget ← 0;
}
```

131. `SD`. ECMA-48 ss. 8.3.113. Scroll Down. `CSI Ps T`.

```
void term_exec_SD(term_xt *tt, action_xt *act)
{
  panel_vscroll(tt→panel, _term_arg(act, 0, 1), IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN));
}
```

132. `SU`. ECMA-48 ss. 8.3.147. Scroll Up. `CSI Ps S`.

```
void term_exec_SU(term_xt *tt, action_xt *act)
{
  panel_vscroll(tt→panel, −_term_arg(act, 0, 1), IS_FLAG(tt→flags, TERM_LEFT_RIGHT_MARGIN));
}
```

133. `VPA`. ECMA-48 ss. 8.3.158. Line Position Absolute. `CSI Ps d`.

```
void term_exec_VPA(term_xt *tt, action_xt *act)
{
  act→arg[0] ← _term_arg(act, 0, 1);
  act→arg[1] ← tt→panel→cursor.col;
  act→narg ← 2;
  term_exec_CUP(tt, act);
}
```

134. `VPB`. ECMA-48 ss. 8.3.159. Line Position Backward. `CSI Ps k`.

```
void term_exec_VPB(term_xt *tt, action_xt *act)
{
  term_exec_CUU(tt, act);
}
```

135. `VPR`. ECMA-48 ss. 8.3.160. Line Position Forward. `CSI Ps e`.

```
void term_exec_VPR(term_xt *tt, action_xt *act)
{
  term_exec_CUD(tt, act);
}
```

136. `XTGETXRES`. `DCS + Q Pt ST`. Xterm uses this to expose its internal state.

```
void term_exec_XTGETXRES(term_xt *tt, action_xt *act)
{
  if (tt→parser.narg) return;
  printf("XTGETRES␣not␣implemented\n");
}
```

137. `XTGETTCAP`. `DCS + q Pt ST`.

```
void term_exec_XTGETTCAP(term_xt *tt, action_xt *act)
{
  if (tt→parser.narg) return;
  printf("XTGETTCAP␣not␣implemented\n");
}
```

138. `XTSETTCAP`. `DCS + p Pt ST`.

```
void term_exec_XTSETTCAP(term_xt *tt, action_xt *act)
{
  if (tt→parser.narg) return;
  printf("XTSETTCAP␣not␣implemented\n");
}
```

139. `XTWINOPS`. X-Terminal Window Operations. `CSI Ps ; Ps ; Ps t` where the first/only argument is less than 24. An initial argument of 24 or above is `DECSLPP`. Treating the list of operations in ctlseqs as authoritative.

```
  void term_exec_XTWINOPS(term_xt *tt, action_xt *act)
  {
    if (tt→parser.narg > 3) return;
    switch (_term_arg(act, 0, 0)) {
    case 1:    /* deiconify */
      if (tt→parser.narg > 1) return;
      break;
    case 2:    /* iconify */
      if (tt→parser.narg > 1) return;
      break;
    case 3:    /* move */
      if (tt→parser.narg > 3) return;
      break;
#if 0
    case 4:    /* resize (pixels) */    /* Omitted parameters reuse the current height or width.
          Zero parameters use the display's height or width. */
      if (tt→parser.narg > 3) return;
      long h ← _term_arg(act, 2, DISPLAY_HEIGHT);
      if (tt→parser.narg < 3) h ← PANEL_HEIGHT;    /* window border? */
      long w ← _term_arg(act, 1, DISPLAY_WIDTH);
      if (tt→parser.narg < 2) h ← PANEL_WIDTH;    /* window border? */
      break;
#endif
    case 5:    /* raise */
      if (tt→parser.narg > 1) return;
      break;
    case 6:    /* lower */
      if (tt→parser.narg > 1) return;
      break;
    case 7:    /* refresh */
      if (tt→parser.narg > 1) return;
      break;
    case 8:    /* resize (cells) */    /* Omitted parameters reuse the current height or width. Zero
          parameters use the display's height or width. */
      if (tt→parser.narg > 3) return;
      break;
    }
  }
```

140. Writing to the host.

```
⟨ Public API (term.o) 5 ⟩ +≡
  int term_write(term_xt *, uint8_t *, size_t);
```

141. ⟨ Function declarations (term.o) 9 ⟩ +≡

```
  static void _term_init_reply(term_xt *, action_xt *, uint8_t, char *, int8_t);
  static void _term_print_reply(term_xt *, action_xt *, int8_t);
  static int _term_print_terminated_string(term_xt *, char *);
  static int _term_printf(term_xt *, char *, . . . );
  static int _term_putc(term_xt *, uint8_t);
  static void _term_reply(term_xt *, action_xt *, int8_t);
  static int _term_vprintf(term_xt *, char *, va_list);
```

142. ⟨ Public API (term.o) 5 ⟩ +≡

```
#define TERM_ANSI_CSI 0x9b
#define TERM_ANSI_DCS 0x90
#define TERM_ANSI_ESC 0x1b
#define TERM_ANSI_OSC 0x9d
#define TERM_ANSI_ST 0x9c
```

143. int *term_write*(term_xt **tt*, uint8_t **buf*, size_t *len*)

```
  {
    if (tt→child_fd ≡ −1) return 0;
    _lio_byte(tt, LIO_OUT, buf, len);
    return write(tt→child_fd, buf, len);
  }
```

144. static void *_term_init_reply*(term_xt **unused tt*, action_xt **act*, uint8_t *mode*, char **prefix*, int8_t *op*)

```
  {
    if (buf_get_length(&act→data)) buf_clear(&act→data);
    memset((char *) act + sizeof (act→data), 0, sizeof (*act) − sizeof (act→data));
    if (prefix) {
      assert(¬prefix[1] ∨ ¬prefix[2]);
      act→prefix[0] ← prefix[0];
      act→prefix[1] ← prefix[1];
    }
    act→mode ← mode;
    act→reply ← op;
  }
```

145. static void *_term_reply*(term_xt **tt*, action_xt **act*, int8_t *op*)

```
  {
    if (op) act→reply ← op;
    if (act ≡ &tt→parser) _term_print_reply(tt, act, act→reply);
  }
```

146. **static void** *_term_print_reply*(**term_xt** **tt*, **action_xt** **act*, **int8_t** *op*)

```
{
  uint8_t mode;
  assert(op);
  mode ← act→mode;
  if ((mode & 0x80) ∧ ¬IS_FLAG(tt→flags, TERM_EIGHTBIT)) {
    _term_putc(tt, TERM_ANSI_ESC);
    mode ← (mode & 0x7f) | 0x40;
  }
  _term_putc(tt, mode);
  if (act→prefix[0]) _term_putc(tt, act→prefix[0]);
  if (act→prefix[1]) _term_putc(tt, act→prefix[1]);
  for (int i ← 0; i < act→narg; i++) {
    _term_printf(tt, "%u", act→arg[i]);
    if (i < act→narg − 1) _term_putc(tt, ';');
  }
  _term_putc(tt, op);
  if (buf_get_length(&act→data)) _term_print_terminated_string(tt, buf_base(&act→data));
}
```

147. **static int** *_term_print_terminated_string*(**term_xt** **tt*, **char** **str*)

```
{
  int r;
  size_t len;
  len ← strlen(str);
  _lio_byte(tt, LIO_OUT, (uint8_t *) str, len);
  r ← write(tt→child_fd, str, len);
  if (r ≡ −1) err(1, "write");
  if ((size_t) r < len) return r;
  if (¬IS_FLAG(tt→flags, TERM_EIGHTBIT)) {
    _term_putc(tt, TERM_ANSI_ESC);
    _term_putc(tt, (TERM_ANSI_ST & 0x7f) | 0x40);
  }
  else _term_putc(tt, TERM_ANSI_ST);
  return r;
}
```

148.
```
static int _term_printf(term_xt *tt, char *fmt, ...)
{
   int ret;
   va_list ap;
   va_start(ap, fmt);
   ret ← _term_vprintf(tt, fmt, ap);
   va_end(ap);
   return (ret);
}
static int _term_vprintf(term_xt *tt, char *fmt, va_list ap)
{
   char lame[10240];
   _lio_byte(tt, LIO_OUT, (uint8_t *) lame, vsnprintf(lame, 10240, fmt, ap));
   assert(tt→child_fd ≠ −1);
   return dprintf(tt→child_fd, "%s", lame);
   return vdprintf(tt→child_fd, fmt, ap);
}
static int _term_putc(term_xt *tt, uint8_t byte)
{
   assert(tt→child_fd ≠ −1);
   _lio_byte(tt, LIO_OUT, &byte, 1);
   return write(tt→child_fd, &byte, 1);
}
```

149. Logging I/O.

```
⟨ Type definitions (term.o) 4 ⟩ +≡
  enum {
    LIO_NONE, LIO_IN, LIO_OUT
  };
```

150. ⟨ Public API (term.o) 5 ⟩ +≡

```
  void term_lio_start(term_xt *, char *);
  void term_lio_stop(term_xt *);
```

151. ⟨ Function declarations (term.o) 9 ⟩ +≡

```
  static void _lio_byte(term_xt *, int, uint8_t *, size_t);
```

152. void *term_lio_start*(term_xt *tt, char *filename)

```
  {
    char defaultfn[PATH_MAX + 1];
    FILE *newfs;
    if (filename ≡ Λ) sprintf((filename ← defaultfn), "turtle.%u.%u.log", getuid(), getpid());
    newfs ← fopen(filename, "w");
    if (¬newfs) return;
    term_lio_stop(tt);
    tt→lio_fs ← newfs;
  }
```

153. void *term_lio_stop*(term_xt *tt)

```
  {
    if (¬tt→lio_fs) return;
    if (fclose(tt→lio_fs) ≡ −1) warn("fclose");
    tt→lio_fs ← Λ;
  }
```

154. static void *_lio_byte*(term_xt *tt, int dir, uint8_t *buf, size_t len)

```
  {
    if (¬tt→lio_fs) return;
    if (dir ≠ tt→lio_dir) {
      assert(LIO_NONE ≡ 0);
      if (tt→lio_dir < 0) tt→lio_dir ← −tt→lio_dir;
      if (tt→lio_dir ≠ LIO_NONE) fprintf(tt→lio_fs, "␣|||\n");
      if (tt→lio_dir ≠ dir) fprintf(tt→lio_fs, (dir ≡ LIO_IN) ? "IN>␣" : "<<<␣");
      else fprintf(tt→lio_fs, (dir ≡ LIO_IN) ? "␣␣>␣" : "␣␣<␣");
      tt→lio_dir ← dir;
      tt→lio_len ← 4;
    }
    for ( ; len; len−−, buf++) {
      if (*buf < 0x20 ∨ *buf ≥ 0x7f) fprintf(tt→lio_fs, "␣%02x", *buf);
      else fprintf(tt→lio_fs, "␣␣%c", *buf);
      tt→lio_len += 3;
      if (tt→lio_len ≥ 68) tt→lio_dir ← −tt→lio_dir;
    }
    fflush(tt→lio_fs);
  }
```

155. Test Terminal.

```
⟨pseudo.c 155⟩ ≡
#include <assert.h>
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <event2/event.h>
#include <X11/keysym.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include "atlas.h"
#include "buf.h"
#include "site.h"
#include "tri.h"
#include "ucpdb.h"
#include "panel.h"
#include "triterm.h"
#include "matrix.c"
#define FONT_NAME "Liberation␣Mono:antialias=true:autohint=true"
#define FONT_SIZE 42
  struct event_base *Loop;
  panel_xt *Panel;
  mat4_t Projection;
  trix_xt *Trix;
  term_xt *Term;
  static void _pseudo_bell(term_xt *);
  static bool _pseudo_dirty(trix_xt *);
  static bool _pseudo_draw_cell(uint32_t, bool, vertex_xt *, size_t, index_xt *, size_t);
  static void _pseudo_gone(term_xt *);
  static void _pseudo_key(XEvent *);
  static void _pseudo_refresh_display(trix_xt *, struct timespec*, bool);
  static void _pseudo_xevent(trix_xt *, XEvent *);
```

See also sections 156, 157, 158, 159, 160, 161, 162, and 163.

156. ⟨pseudo.c 155⟩ +≡

```
int main(unused int argc, unused char **argv)
{
  setlocale(LC_CTYPE, "");
  Loop ← event_base_new();
  if (¬Loop) errx(1, "event_base_new");
  do_tri_init(Trix, Loop, _pseudo_xevent, 640, 480);
  if (¬Trix) errx(1, "tri_init");
  else if (Trix ≡ (trix_xt *) −1) return 0;
  tri_set_draw_fun(Trix, _pseudo_refresh_display);
  tri_set_xevent_mask(Trix, KeyPressMask);
  Panel ← new_panel(80, 24, _pseudo_draw_cell);
  if (¬Panel) errx(1, "new_panel");
  panel_load_xft_font(Panel, tri_xdisplay(Trix), tri_xscreen(Trix), FONT_NAME, FONT_SIZE);
  panel_begin_cursor_blink(Panel);
  tri_set_query_dirty_fun(Trix, _pseudo_dirty);
  Term ← new_term(Loop, Panel);
  if (¬Term) errx(1, "new_term");
  if (¬term_attach_pty_exec(Term, "/usr/local/bin/vttest", Λ))
    errx(1, "term_attach_pty_exec");
  term_install_signal_handler(Term);
  term_set_reap_fun(Term, _pseudo_gone);
  term_set_bell_fun(Term, _pseudo_bell);
  term_set_speed(Term, 115200, 0.1);
  XMapWindow(tri_xdisplay(Trix), tri_xwindow(Trix));
  tri_go(Trix);
  term_destroy(Term);
}
```

157. ⟨pseudo.c 155⟩ +≡

```
static bool _pseudo_dirty(trix_xt *t)
{
  assert(t ≡ Trix);
  return 1 ∨ panel_get_dirty(Panel);    /* cursor moving not taken into account */
}
```

158. ⟨pseudo.c 155⟩ +≡

```
static void _pseudo_gone(term_xt *tt)
{
  assert(tt ≡ Term);
  assert(¬term_has_pty(Term));
  tri_destroy(Trix);
}
```

159. ⟨pseudo.c 155⟩ +≡

```
static void _pseudo_bell(term_xt *tt)
{
  assert(tt ≡ Term);
  printf("Ding␣dong!\n");
}
```

160. ⟨ pseudo.c 155 ⟩ +≡

```
static void _pseudo_xevent(trix_xt *t, XEvent *ep)
{
  assert(t ≡ Trix);
  switch (ep→type) {
  default: fprintf(stderr, "unhandled␣event␣%d\n", ep→type);
  case ConfigureNotify: case Expose: case MapNotify: case UnmapNotify: case ReparentNotify:
    break;
  case 0:
    if (term_has_pty(Term)) term_close_pty(Term);
    break;
  case KeyPress: _pseudo_key(ep);
    break;
  }
}
```

161. ⟨ pseudo.c 155 ⟩ +≡

```
static void _pseudo_refresh_display(trix_xt *t, struct timespec* at, bool filthy)
{
  size_t s[2];
  assert(t ≡ Trix);
  panel_get_pixel_size(Panel, s);     /* Changes with font */
  Projection ← m4_ortho(0, s[0], 0, s[1], −1, 1);
  if (¬panel_redraw(Panel, at, filthy)) fprintf(stderr, "draw␣error\n");
}
```

162. ⟨ pseudo.c 155 ⟩ +≡

```
static bool _pseudo_draw_cell(uint32_t t, bool strip, vertex_xt *v, size_t vl, index_xt *i, size_t
        il)
{
  return tri_draw_vertices(Trix, strip ? GL_TRIANGLE_STRIP : GL_TRIANGLES,
      (float *) &Projection, t, v, vl, i, il);
}
```

163. The X.Org implementation of *XLookupString* in `KeyBind.c` either copies the string a keysym has been rebound to into *buf* or makes a stab at guessing which character is represented and puts that in the first byte of *buf*. In neither case is any effort made to ensure the string is terminated with a zero.

However, the only strings longer than one character (the low 7 or 8 bits of the keysym) that could be returned by *XLookupString* are those already provided by this client to *XRebindKeysym*.

Xutf8LookupString boils away to *_XimLookupUTF8Text* in `imConv.c` which *doesn't* correctly check that *nbytes* is one less than its `BUF_SIZE` (20).

In fact the *XLookupString* from *XKBBind.c* is used. This ends up in *XkbTranslateKeySym* where Λ is replaced with its own 4-byte buffer and it's the length of what's written in there that's returned.

Why not 5? Maximum encoded length is 4 + terminator?

```
⟨pseudo.c 155⟩ +≡
  static void _pseudo_key(XEvent *ep)
  {
    KeySym k;
    int len;
    char buf[8];
    len ← XLookupString(&ep→xkey, buf, 8, &k, Λ);
    printf("%06lx␣--␣%u␣bytes%s\n", k, len, len ? ":" : "");
    if (len ≡ 1 ∧ k ≡ XK_BackSpace) term_write(Term, (uint8_t *) "\177", 1);
    else if (len) term_write(Term, (uint8_t *) buf, len);
    else
      switch (k) {
      case XK_Left: printf("left\n");
        break;
      case XK_Up: printf("up\n");
        break;
      case XK_Right: printf("right\n");
        break;
      case XK_Down: printf("down\n");
        break;
      default: printf("unexpected␣key␣0x%lx\n", k);
      }
  }
```

164. Standards Notes.

Terminals have been in development for a long time.

Turtle implements some of ECMA-48 and DEC STD 070.

Terminfo describes the capabilities that a terminal has in a machine-accessible form. A lot of functionality described by ECMA and DEC is assumed by terminal-using applications. Some of the things they standardise were in flux at the time and terminfo describes their implementation as well as extensions to the standard.

ISO-646 is ECMA-6: 7 bit character set (ie. ASCII) ISO-2022 is ECMA-35, ANSI-X3.41: 7 & 8 bit code extension techniques ISO-4873 is ECMA-43: 8 bit companion of ISO-646 ISO-6429 in ECMA-48, ANSI-X3.64: control functions ISO-7498 is the OSI network model ISO-10646 is unicode

165. ECMA-48. Section 8.3 defines the control codes. Many of these are obsolete in modern terminals. Controls which are obsolete or not implemented are not listed unless they are notable.

Implementation is mostly complete except tabulation and some cursor movement commands.

ECMA-48 has 4 (?) ways to move the cursor down a line: CNL, CUD, NEL, LF.

TODO: tabulate.

`APC` Application Program Command. Parsed and ignored.

8.3.3 `BEL`. Implemented.

`BPH` Break Permitted Here. Obsolete.

Superceded by unicode.

8.3.5 `BS` Backspace.

8.3.6 `CAN` Cancel.

`CNL` Cursor Next Line. `NEL` in DEC STD 070.

8.3.14 `CPR` Active Position Report.

8.3.15 `CR` Carriage Return.

Mentions features which have been replaced with windowing, itself now largely obsolete.

`CSI` Control Sequence Introducer.

8.3.18 `CUB` Cursor Left.

8.3.19 `CUD` Cursor Down.

8.3.20 `CUF` Cursor Right.

8.3.21 `CUP` Cursor Position.

8.3.22 `CUU` Cursor Up.

8.3.24 `DA` Device Attributes. Parsed. Partially implemented.

`DCS` Device Control String. Parsed and ignored.

8.3.39 `ED` Erase in Page.

`ESC` Escape. Parsed.

Many sequences unimplemented.

8.3.26 `DCH` Delete Character.

8.3.32 `DL` Delete Line.

8.3.38 `ECH` Erase Character.

8.3.51 `FF` Form Feed.

8.3.60 `HT` Character Tabulation. Not implemented.

8.3.62 `HTS` Character Tablulation Set. Not implemented.

8.3.63 `HVP` Character and Line Position.

8.3.64 `ICH` Insert Character.

8.3.67 `IL` Insert Line.

8.3.74 `LF` Line Feed.

Mentions features which have been replaced with windowing, itself now largely obsolete.

8.3.86 `NEL` Next Line.

8.3.88 `NUL` Null. Implemented and ignored.

`OSC` Operating System Command. Parsed and ignored.

8.3.104 `RI` Reverse Line Feed.

`SD` Scroll Down. Not implemented.

Originally moved (scrolled) the displayed data. Soon replaced by Index which moves the cursor down "without moving horizontally".

`SGR` Select Graphic Rendition. Not implemented.

`SM` Set Mode.

Not implemented even to modern standards. Only four modes mentioned by ECMA are listed in ctlseqs.txt, including `20` ("mode two-zero", not twenty modes) which ECMA says "shall not be used".

8.3.156 `TBC` Tabulation Clear.

8.3.161 `VT` Line Tabulation.

166. DEC. DEC-STD-070 is not a public standard but a corporate document describing how Digital's VT line of terminals should work.

1.5 Conformance.

A level is a superset of the previous level and new functionality.

Each level consists of functions which must be included (required).

Conformance to level n:

SHALL implement ALL functions 1..n.

MAY NOT implement functions of level ¿ n.

Some vt100/vt125 (vt52?) functions MAY NOT be included.

Level 1: vt102, vt125.

Not planned: Shifting, Character sets.

Not planned yet: SGR, double-size.

Not discovered in docs: DECSC, DECRC.

Level 1 extensions:

What to do about Insert/Replace Mode?

Not planned: 132-column mode, printer port, katakana.

Not planned yet: ReGIS, Sixel.

Level 1 preferences:

scrolling, reverse-video, auto-repeat.

Level 2.

Not planned: shifting, character sets.

Level 2 extensions:

Not planned: dynamic character sets, user defined keys.

3.1.2 NRCS National Replacement Character Set.

A set of 7 bit extensions to the vt100 introduced with the vt200. By 1989 these have been deprecated in favour of 8 bit multinational character sets which they have subsequently been superceced by unicode.

3.1.3 8-bit Interface Architecture.

This is also largely obsolete in favour of unicode but not quite as much as the 7 bit raplacement character sets.

3.1.4 Seems to describe how the 8-bit interface interfaces with the Terminal Interface Architecture, described briefly in appendix B, which seems to more or less describe the curses library.

B.3 GL is ASCII, GR is DEC Supplemental. "That assumption should be adhered to by all conforming software unless some very explicit, well-documented additional mechanisms are used". Locking shifts are explicitly called out as being for exceptional cases.

B.4.1.1 Let a lower level do argument compression on the line.

B.4.1.2 16 parameters (recommended maximum) with values up to at least (minimum maximum) 255 are supported universally. Larger bounds may not be.

B.4.2 The line is not in a known state when it's initialised.

This should be irrelevant for turtle even if put on a physical line but here is the summary for completion:

B.4.2.1 Restore communication with XON.

For some reason this was not included in the full sequence below.

B.4.2.2 Terminate any existing control string with ST.

B.4.2.2.1 Printer Control Mode. Obsolete form of directing output to the attached printer instead.

B.4.2.2.2 Select Conformance Level specifices the "compliance level" to DEC's specification. How (or if) this implementation will map to DEC conformance levels is not clear.

B.4.2.3 Full sequence: 7-bit ESC ESC [4 i ESC [6 2 " p 8-bit ST CSI 4 i CSI 6 2 " p

B.4.2.4 Device Control String. States that control strings should be accepted but not acted upon in a DCS (in fact it states only BS, CR and LF need be "recognised and interpreted" within the control string and calls out some implementations which may act on other control characters differently. This implementation does not make such a distinction and copies the 8-bit device control string as-is.

APC, OSC and PM strings are also mentioned but as not implemented by any DEC product and "should not be used".

B.4.2.5 Select 7-bit and 8-bit C1 Transmission. Described in detail in the "Code Extension Layer" chapter, this should have been specified by the conformance layer (CSI 6 2 " p means level 2: vt200).

Turtle will implement this feature and allow 8-bit C1 control sequences but it should probably not be used. This only governs how C1 controls are sent to the host: the terminal will always accept 7-bit and 8-bit C1 controls.

B.5.2.1 "The first terminal function to be performed by [the operating system] is the Primary Device Attributes request".

Ironically: "because devices are identified to the operating system according to functional, rather than product, characteristics". DA1 focuses almost entirely on product.

B.5.2.2 DA2. Device (product) identification. For backwards compatibility.

B.5.2.3 DECID deprecated. Only implemented at level 1 or 2 conformance?

B.5.2.4 DECSCL allows the application to set terminal conformance level. The recommendation is to leave the terminal at its higher conformance level and emulate the lower level in the operating system.

B.5.3 Device test is "strictly for hardware diagnostic".

B.5.4 For some reason this is the place to mention that the cursor graphic can be turned off.

B.6 Local functions (and their emulation) not revelant. This feature has eventually become the cooked/cbreak/raw distinction performed by the kernel pty driver.

B.7.1 Auto Repeat. Recommends to handle it as-is and not disable it, which is convenient because X.

B.7.2 Typing Ahead. No longer relevant.

B.8 ReGIS. TODO much later.

3.3 describes ASCII, how to fit ASCII into 8 bits, and how to handle control codes. This is realised as the parser taken from vt100.net.

3.4 How to treat space, delete and their GR equivalents. Delete refers to DEC STD 070-6.

3.5 General description of escape sequences, control sequences and control strings. This is, again, embodied in the FSM from vt100.net.

3.5.1.2.1 CAN terminates immediately "without execution", any sequence in progress.

3.5.1.2.2 SUB can be seen as a historical analog of the unicode replacement character but the effect now is identical to CAN.

3.5.1.2.3 ESC cancels any escape or control sequence and starts a new one.

3.5.1.2.4 C1 controls. Implied by 3.5.1.2.3 because C1 controls start with ESC in a 7-bit environment.

3.5.1.2.5 ST declared universal. Terminates:

a Escape sequences in progress (implied by 3.5.1.2.3). b Control sequences in progress (— „ —). c Control string in progress. May be normal string termination or a variant of 3.5.1.2.5.b. d Nothing (ie. ignored as usual if nothing is in progress).

A note adds: products may implement non-ANSI modes that do not recognise ST as documented exceptions.

3.5.1.3 Unimplemented functions ignored as if not received. Applies to controls and parameters.

Turtle is currently too strict (but only complains on its stdout).

3.5.1.4 Output routines need more control over the data (using write(2) is too crude) to properly care for 7/8-bit C1/G2/G3 controls.

0xff explicitly called out to be ignored in sequences and other 8-bit codes are treated as equivalent 7-bit code.

SI and SO can occur in 7-bit environment: within a CSI, ST-terminated string or between SS and its character. Will terminal implement shifts?

3.5.2 ESC sequence is ESC, 0 or more intermediates from 0x2x, a final from 0x3x to 0x7x excluding 0x7f.

"At least three" intermediates should be accepted. More should be detected and the sequence ignored.

3.5.3 CSI. Prefix characters are not mentioned here but intermediates after the parameters are.

"At least three" intermediates should be accepted. More should be detected and the sequence ignored.

3.5.3.1 Recommends minimum of 14 bits (16384) for parameter values. Larger values SHOULD BE mapped to the largest.

Up to 16 parameters is required for compatibility. More MAY BE ignored.

Semicolon to separate paremeters. Colon reserved for future standardisation but if received the sequence should be accepted and ignored. 0x3a–0x3f described in 3.5.3.4. Leading 0's ignored. 0 or absent represents a default.

ISO 6429 stamps all over that an allows 0 to be a value with knobs on for compatibility. DEC decided bugger that but did it anyway on a few things (see DEC STD 138-0 but I'm going to gamble that it doesn't matter).

3.5.3.2 States again that 0 or an absence is the default value to "one parameter or to the entire parameter string" when the paremeters are numeric.

3.5.3.4 0x3c-0x3f in the first position means the whole string is subject to "special interpretation" not defined in any standard. 0x3c-0x3f anywhere else causes the sequence to be ignored.

3.5.4 Control *Strings*. As opposed to control sequences. Introduced by APC, DCS, OSC or PM. How C0 control are interpreted is up to each mode but they are not acted upon.

Thanks to Linux controls other than ST must be used to detect string termination.

3.5.4.3 Adds SOS to the list of control strings to understand and ignore.

Also notes that ISO may define escape and control sequences inside Character Strings (to the best of my knowlege it didn't).

3.5.4.4 ST but also "0x08–0x0d are valid within control strings".

Other controls than ST will terminate strings mostly for safety reasons but "conforming software should not depend on this practice" and notes that other control characters "may be defined to have other meanings in the future". The recommendation is to ignore all other controls.

The linux people did not get that memo.

3.5.4.5 Ignore the 8th bit in control strings that are going to be ignored anyway (APC, OSC, PM). Let the consumer decide what to do with them in DCS strings *except in the DCS introducer sequence* where the 8th bit should also be ignored.

Turtle does not do this correctly because I had to guess (TODO: fix).

3.5.4.6 Parse but ignore control strings.

3.5.5 An example of all of the above in VAX-11 C, dated 1984–1989.

3.6 Character sets. 8-bit has two graphic regions, GL and GR. Terminal has four sets of graphic characters, G0, G1, G2 and G3. Character sets may be designated to any of the four sets and each set may be invoked onto GL or GR, by *shifting*.

This functionality has been largely or entirely superceded by unicode.

LS0–LS3 lock the corresponding G-table into GL (LS0 and LS1 were previously Shift In and Shift Out but the meaning has not changed).

LS1R–LS2R lock the corresponding G-table into GR.

Unaffected by shifting:

C0 and C1 controls. SP and DEL or 0xa0/0xff with GL or GR has a 94-character set. Contents of an escape sequence. The byte following SS2 or SS3.

Speaking of, SS2 and SS3 (Single Shift) invoke G2 or G3 into GL.

G0 cannot have a 96-character set (SP and DEL always SP and DEL).

3.6.3 User Preference Supplemental Set (UPSS).

Superceded by Private Use Areas in unicode. Support is required for conformance level 3 at least to switch between ISO Latin-1 and DEC Supplemental. Refers to DEC STD 070-6.

3.6.4.1 Level 1, G0-G3 are all ASCII. G0 into GL. 7 bits only.

3.6.4.2 Level 2 (without 8-bit extension). ASCII in G0, G1. DEC Supplemental in G2, G3. G0 in GL, G2 in GR.

3.6.4.3 Level 3 (or level 2 with 8-bit). ASCII in G0, G1. UPSS in G2, G3. G0 in GL, G2 in GR.

Level 3 also recognises a reset from ISO-4873.

3.6.4.4 NRCS (national replacement). Shifting may be necessary/useful for graphics but this one's definitely taken over by unicode.

3.8 Specifies how to indicate conformance to ISO-4873 (ECMA-43) which extends the Gn system in ways we probably don't need. Note that ISO-4873 levels (1–3) are different from DEC conformance levels (1–3).

3.8.2 S7C1T/S8C1T Select [78]-bit C1 Transmission. Needed. Defaults to 7-bit.

3.8.3 Shifting. SI/LS0, SO/LS1, LS[23], LS[123]R, SS[23]

4.3 State Descriptions.

4.4 Device Initialisation.

4.5 Control Functions (DA).

Has list of extensions to return in DA1 (CSI Ps c) on pp. 4-19.

4.5.X DECSCL Select Conformance Level.

4.5.X DA2 Request/report device identification.

4.5.X DA3 Request terminal unit identification. Xterm sends zeros.

4.5.DECID Request device identification code.

4.5.DECSR Secure Reset. May be useful after password input. Not in ctlseqs (thus not in Xterm?).

4.5.DECSTR Soft Terminal Reset.

5.4.1 Logical Display. 80x24 or 132x24. Starts at 1x1 in top left.

No windows extensions.

Glyph location corresponds to lower-left corner of co-ordinates (this matters for double width/height characters).

5.4.2 DECTCM Text Cursor Mode (terminfo civis, cvvis, cnorm).

5.4.3 Margins/Scrolling.

5.4.3.DECSTBM, Set Top and Bottom Margins. Terminfo csr.

5.4.3.DECSLRM, Set Left and Right Margins. Terminfo smglp, smglr, smgrp.

5.4.3.DECLRMM, Left/Right Margin Mode. Terminfo mgc.

5.4.3.DECOM, Origin Mode. Not used by Xterm terminfo but part of DECSTR.

5.4.3.ECSCLM, Scrolling Mode.

5.4.3.IND. Move down with scrolling. Ignores New Line Mode unlike LF (except on vt100 and vt125, where they're identical).

5.4.3.RI. Reverse index. Unavailable on vt100 or vt125 or non-quirky?

5.4.3.DECFI. Forward (right) Index.

5.4.3.DECBI. Backward (left) Index.

5.4.4 Cursor Movement.

CUB/CUD/CUF/CUP/CUU, CR, LF from ECMA-48.

5.4.4.CPR, Cursor Position Report. Part of u6/u7/u8/u9. According to Xterm terminfo for tack and/or ANSI ENQ.

5.4.4.DECXCPR, Extended Cursor Position Report.

5.4.4.LNM, New Line Mode.

Whether receipr of LF, VT and FF do their own thing (not set) or also do CR (set). Also return key will *send* CR LF.

Deprecated (should not be used == off?) for conforming software.

Not in Xterm terminfo.

5.4.4.VT, Vertical Tab. Useful on line printers. LF on screen.

5.4.4.FF, Form Feed. Likewise.

5.4.4.BS, Back Space. Unaffected by Auto Wrap Mode. Cursor never advances to previous line.

Erasure, if done, is not done here.

5.4.4.NEL, Next Line. Explicitly LF in New Line Mode or CR LF.

5.4.5 Tabs. TBD.

167. Terminfo.

The terminfo file describes which features or bugs a terminal has and how to activate certain features. Applications use the terminfo database and also make assumptions that the terminal is some degree of ECMA-48, DEC VT and xterm compatible.

The description of the terminfo file that accompanies this terminal continues in `turtle.info`.

168. ctlseqs.

Documentation on controls comes largely from https://invisible-island.net/xterm/ctlseqs/ctlseqs.txt

169. Unicode.
(Summary of 5000 years of writing and all the damage it's caused)

170. Keyboard.

This is exclusively from DEC-STD-070 ss. 6. Most is obsoleted by X and/or unicode.

Two modes: typewriter, data processing (switches between keycaps).

See also DEC-STD-107 (level 1). DEC-STD-169 level 2 adds character sets, keypad and function keys. Level 3 doesn't add a great deal.

Level 1 conformance:

All 1 keys, no other, 2 or 3 keys. DECARM, DECCKM, DECKPAM/DECKPNM, Line Feed/New Line Mode. KAM (Keyboard Action Mode) Caps lock, maybe shift lock. Enable 8 bit transmission.

Level 2 conformance:

All 2 keys, no others. Optional 8 bit transmission. Optional NRCS. L1 controls KAM + DECSCL Keyclick, caps/shift lock (caps lock by factory default!), optional bell, etc.

Level 3 conformance:

Level 2 plus: 8 bit required. DECKBUM DECSCL again Optional composing keys

Level 1 op:

No 8th bit, or 8th bit unset. No E1-E6. No F6-F10. F11 =¿ ESC 0x1b F12 =¿ BS F13 =¿ LF No F14-F20. Really no 8th bit (beep if the user tries!)

Level 2/3 op:

All keys. F11-13 transmit as usual. All valid compose sequences OK ”No additional Compose sequences”

But in a 7-bit environment: No composing or transmission with 8th bit.

”Delete” where PC has Backspace (but with the same long left arrow) Caps lock is two keys: ctrl and lock. 20 F-keys (not called F). Keypad has four function keys, normal height [-,] in place of +. F1 == Hold F2 == Print F3 == Setup F4 F5 == Break F11 == ESC F12 == BS F13 == LF

Modes: KAM, Repeat, Click, Margin bell, Warning bell.

9 keystroke (not byte) “silo” (buffer).

DEC-STD-070 ss. 6.6.2, DEC-STD-070 p. 6-38 Keyboard Action Mode locks the keyboard on the SM control or a full silo.

Locked keyboard: wait indicator All but local keys (F1-F5) disabled F5/break may be disabled elsewise Unlocks for set-up (unless unlocked by the user)

Unlocks for: silo space (if not KAM) RM KAM Select conformance level (DECSCL) Reset (RIS) Self test Enter Set-Up in level 1 (doesn't clear in other levels) In Set-Up: Clear Comm or Reset Terminal Soft reset (DECSTR)

KAM Keyboard Action Mode CSI 2 [hl].

DEC-STD-070 ss. 6.6.3, DEC-STD-070 p. 6-40. DECARM Auto Repeat Mode. CSI ? 8 [hl].

Non-repeating keys: Return, Escape, F1-5, Ctrl, Shift, Lock, Compose, diacrits, 'in a ... compose sequence'.

Repeats different keys at different rates and depending on connection speed (baud rate).

6.6.4.1 Visual indicator of HOLD SCREEN (XON/XOFF?).

6.6.4.2 — „ — lock key

6.6.4.3 — „ — compose key

6.6.4.4 Keyboard Lock (silo full). This *does* mention XOFF. Locks if mid-compose.

6.6.5 Bells. Warning, Margin.

Warning max 4-5 per second. Bells queued until buffer (10) is full then discarded.

The margin bell rings when moving to within 8 characters of the right margin. This might be an interesting anachronism but useless today.

6.6.5.3 Keyclick. Once per repeat. Warning bell takes precedence.

6.7 Affecting modes:

Emulation (of vt52, vt100, vt300). C1 transmission Character set (keyboard encoding) UPSS user graphic character set Keyboard Dialect Keyboard Usage (data/typewriter) Host (line) environment. Key lock BackArrow key mode (not sure when this is from — 1988? Already disagreements about BS/DEL). Disable break. Disable compose key. Various other things replaced by xkeymap. Cursor Key Mode: ANSI vs. Application Function Codes. Numeric Keypad Mode: Numeric vs. Application Mode.

`S7C1T`, `S8C1T`. DEC-STD-070 ss. 6.7.3, DEC-STD-070 p. 6-50. Select [78]-bit C1 Transmission. `ESC SP F` (7), `ESC SP G` (8).

TODO: Why bytes 6 and 7 (F, G) not 7 and 8?

6.7.4 Character Set Mode selects between 7- and permitted to send 8-bit characters. May be where we put the unicode toggle, if it's toggleable. CSI ? 4 2 [hl]. h=7, l=8.

6.7.5 Keyboard Usage Mode. ie. Local language. Another good choice for the unicode toggle.

DECKBUM (data/typewriter) described on DEC-STD-070 p. 6-57.

6.7.6 Keyboard Dialect. Another local language control. Another choice for unicode.

6.8.1 Cursor Key Mode (how to transmit cursor keys). DECCKM.

6.9.1 — „ — keypad. DECKPNM. DECKPAM. DECNKM.

6.11 Application Function Keys. Lists codes to send for function keys (here defines codes ending ␣). Only lists F6-F20.

6.12 Local Function Keys. See DEC-STD-070 ss. D now that there's no such thing.

6.12.1 Hold Screen Key. Different from XON/XOFF. See DEC-STD-070 ss. 12.

6.12.2 Print Screen Key. Unimplemented for now. See DEC-STD-070 ss. 7.

6.12.3 Set-Up Key. Unimplemented. Covered by configuration, CLI and a GUI/API.

6.12.4 More fancy keys not useful in X.

6.12.5 Break key. Very magic. Refers to DEC-STD-070 ss. 3, DEC-STD-070 ss. 4 and DEC-STD-070 ss. 12. Probably useless in X.

6.13 Main key array — specials. X.

6.13.4 Ctrl-space will send a NUL.

6.13.5 Return sends CRLF if NLM. Much detail here for emulation.

6.13.7 Delete key usually causes 0x7f, also aborts compose. VT300 permits Set-Up to toggle `<X]` between 0x08 and 0x7f.

Trivia: This was a settable option on the VT320 released in 1987. The product was obviously in development long before that and the argument itself must have been fairly old hat by then. Linux was released in 1991. It did not originate the argument.

6.14 Maaaany pages of keyboard layouts.

6.15 Maaaany pages of compose sequences.

6.16 Specifies control keys to sequences, alphabet plus: `^SP`: 00 `^2`: 00 `^3`: 1b (after z: 1a) `^[`: 1b `^4`: 1c `^\`: 1c `^5`: 1d `^]`: 1d `^6`: 1e `^␣`: 1e `^7`: 1f `^?`: 1f `^8`: 7f `^TAB`: 09 `^DET`: 18

`^Q`, `^S` only if XOFF handling is disabled.

Shift ignored.

6.17 Summary.

More from xterm faq F1–F4 & PF1–PF4, ESC O P–S No break key (sends "break") All functions are ESC O x for any x or CSI Pn .

171. Index.

TERM