guile-termios(3) Manual

Frank Terbeck

October 12, 2014

Contents

1 NAME

`guile-termios` - POSIX Termios interface for GNU Guile Scheme

2 SYNOPSIS

```
;; The basic interface:
(use-modules (termios))
;; A similar interface, that throws exceptions in
;; case of errors from the C library procedures:
(use-modules (termios with-exceptions))
;; System dependent constants and structures:
(use-modules (termios system))

;; Sample code:
(define tty (open-io-file "/dev/ttyUSB0"))
(define ts (make-termios-struct))
(tc-get-attr! tty ts)
(cf-make-raw! ts)
(tc-set-attr tty ts)
(close-port tty)
```

3 DESCRIPTION

To query and change settings of serial devices on POSIX systems, the termios API is used. This module implements access to these POSIX functionality for GNU Guile scheme by use of its dynamic foreign function interface.

In order to do this, the module provides conversion between `struct termios` instances from C land to a corresponding Scheme structure and utilities to work with said Scheme structure.

In order to improve portability (the way termios is implemented differs from system to system) the module uses a helper program implemented in C during build time to generate the contents of the `(termios system)` module. Most importantly, this module contains termios constants such as `B9600`. The module actually prefixes all constants derived from `#defines` by `termios-`, so that example constant is actually available as `termios-B9600`.

The library offers two interfaces: `(termios)` and `(termios with-exceptions)`. The latter offers the same access as the former, but automatically raises errors when one of the C-library procedures signals failure.

This manual describes version `0.4` of the guile-termios library.

4 API

The described API is offered by both `(termios)` and `(termios with-exceptions)`, almost completely. See the *"The (termios with-exceptions) module"* section below for details about the differences.

4.1 `struct termios` Handling

`(make-termios-struct [scheme-structure])` Return a `struct termios` instance. If *scheme-structure* is provides, use it to determine the contents of the instance. If it is not provided, the instance is filled with zeroed out values.

`(parse-termios-struct termios-struct)` Take a `struct termios` instance from C-land and return a corresponding Scheme structure. You should *not* modify that structure manually, but rather use the `get-field-from-termios` and `put-field-into-termios!` API functions.

`(get-field-from-termios scm field)` Extract `field` from `scm` (the Scheme structure corresponding to a `struct termios` instance). Valid fields are actually all fields of `struct termios` from `termios.h`; but you should probably only modify *c-cflag*, *c-iflag c-lflag* and *c-oflag* unless you really know what you are doing. See *termios(3)* for details about the structure's fields and their values.

`(put-field-into-termios! scm field value)` Put `value` into `field` of `scm` (the Scheme structure corresponding to a `struct termios` instance). See `get-field-from-termios` about possible fields and their values.

4.2 The `(termios)` module

All of these procedures work on `struct termios` instances from C-land, since they wrap the corresponding C-library procedures. This reference uses `ts` for these parameters. Also instead of working with file-discriptor integers, like the C-library does, the Scheme interface uses *ports*, which is the standard file access interface in GNU Guile Scheme. Parameters of this kind are referenced as `port`.

The C-library procedures signal errors via return values and offers diagnostic information via the POSIX `errno` mechanism. Currently, GNU Guile's dynamic FFI does not feature portable access to that value. Therefore, this module offers access to `errno` in order to enable useful diagnostics. See the *"Access to errno"* section below for details.

`termios-version` This is a string constant, that describes the version of the guile termios module.

`(termios-failure? int)` Returns either `#t` or `#f`, depending on whether or not `int` signals an error returned by one of the C-library procedures wrapped by the module's API. All values returned by procedures marked as **<failure?>** may be tested using this function.

`(tc-get-attr! port ts)` **<failure?>** Get the current attributes from the serial device referenced by `port` and save the retrieved values to `ts`. *This procedure wraps the* `tcgetattr` *procedure from the C-library.*

`(tc-set-attr port ts [#:when action])` <**failure?**> Take the termial attributs from `ts` and apply them to the serial device referenced by `port`. The `action` parameter defines when this action is to take place; it defaults to `termios-TCSANOW` (see *termios(3)* for details). *This procedure wraps the* `tcsetattr` *procedure from the C-library.*

`(tc-drain port)` <**failure?**> `tc-drain` waits until all output written to the serial device referenced by `port` has been transmitted. *This procedure wraps the* `tcdrain` *procedure from the C-library.*

`(tc-flow port action)` <**failure?**> Depending on the value of `action` (see *termios(3)* for details), this suspends the transmission or reception of data to or from the serial device referenced by `port`. *This procedure wraps the* `tcflow` *procedure from the C-library.*

`(tc-flush port queue-selector)` <**failure?**> `tc-flush` discards data written to the serial device referenced by `port` but not transmitted, or data received but not read, depending on the value of `queue-selector`. See *termios(3)* for details. *This procedure wraps the* `tcflush` *procedure from the C-library.*

`(tc-send-break port duration)` <**failure?**> `tc-send-break` transmits a continuous stream of zero bits for a specified `duration`, if the device referenced by `port` is an asynchronous serial device. For details about the `duration` parameter see *termios(3). This procedure wraps the* `tcsendbreak` *procedure from the C-library.*

`(cf-make-raw! ts)` <**void**> Sets the attributs in `ts` to a "raw" mode. *This procedure wraps the* `cfmakeraw` *procedure from the C-library.* That procedure is an extension to the POSIX standard. If the procedure is not available, the module provides a fallback using the `cfmakeraw-fallback` function. For that reason `cf-make-raw!` is implemented as a module.

`(cf-get-ispeed ts)` <**baudrate**> Extract the incoming speed value from `ts`. This value is one of the `B*` constants from the *termios.h* header file. *This procedure wraps the* `cfgetispeed` *procedure from the C-library.*

`(cf-get-ospeed ts)` <**baudrate**> Extract the outgoing speed value from `ts`. This value is one of the `B*` constants from the *termios.h* header file. *This procedure wraps the* `cfgetospeed` *procedure from the C-library.*

`(cf-set-ispeed ts speed)` <**failure?**> Set the incoming speed value in `ts` to `speed`. This value is one of the `B*` constants from the *termios.h* header file. *This procedure wraps the* `cfsetispeed` *procedure from the C-library.*

`(cf-set-ospeed ts speed)` <**failure?**> Set the outgoing speed value in `ts` to `speed`. This value is one of the `B*` constants from the *termios.h* header file. *This procedure wraps the* `cfsetospeed` *procedure from the C-library.*

`(cf-set-speed ts speed)` <**failure?**> Set both the incoming and the outgoing speed in `ts` to `speed`. *This procedure wraps the* `tcsetattr` *procedure from the C-library.* That procedure is an extension to the POSIX standard. If it is not available in the C-library, the module provides a fallback implemented using `cf-set-ispeed` and `cf-set-ospeed`.

4.3 Access to `errno`

`errno` is an integer value that refers to a reason for C-library procedure to fail. It is significant when the library function's return value signals that an error has happened. The way `errno` is implemented differs from system to system, since some systems implement per-thread errno locations, which is usually achieved by a C preprocessor macro along the lines of this:

```
#define errno (*__errno())
```

This library implements the following procedure to implement access to `errno`:

`(get-errno)` Return the current value of `errno`.

This is to be used right after the return value of one of the termios procedures was examined (*do* **not** *do it like this!*):

```
(if (termios-failure? (tc-drain port))
    (let ((errno (get-errno)))
      ...))
```

The problem with this is, that Guile's runtime may run C-library procedures that touch `errno` in between the `tc-drain` and `get-errno` calls. To avoid this, `call-with-blocked-asyncs` should be used. In order to ease the use of this, the `(termios)` module offers a `(call-with-errno ...)` form:

`(call-with-errno (var expression) fail-expressions ...)` This form evaluates `expression` and stores the return value in a variable named `var` for later reference. In case `(termios-failure? var)` returns `#t`, the expressions in the form's body are evaluated, with the value of the last expression being the return value of the `call-with-errno` form. In case `var` did not signal an error, the form evaluates to `#t`.

The *"Examples"* section below features an example of how to use `call-with-errno`.

4.4 The `(termios with-exceptions)` module

This module offers another approach to the same POSIX functionality as the `(termios)` module does. The difference being, that when any of the procedures that may fail signal failure, this module raises an error. Possible exceptions, raised by this modules are:

`termios/no-such-field` Raised by field accessors for the Scheme representation of `struct termios` instances (that means either `get-field-from-termios` or `put-field-into-termios!`) in case the supplied `field` parameter is *not* a valid field name within the termios structure.

`system-error` Raised by any of the termios procedures, that may fail in case of an actual failure. The error-handling of the procedures of this module properly access `errno` using `call-with-errno` and offer the raw value as well as a human readable string (retrieved via `strerror`) as arguments to the exception.

The API of the `(termios with-exceptions)` module otherwise exactly matches the one of the basic `(termios)` module, except that it *does not* export the `errno` access API, since the error handling of the module takes care of that for the user already.

5 EXAMPLES

In the first example, let's open a serial device and set it to `9600bd 8N1` mode. The example uses the `(termios with-exceptions)` module, which means that any failure would cause the script to error out automatically:

```
(use-modules (termios with-exceptions)
             (termios system))
(define tty (open-io-file "/dev/ttyUSB0"))
(define ts (make-termios-struct))
(tc-get-attr! tty ts)
(cf-make-raw! ts)
(cf-set-speed ts termios-B9600)
(tc-set-attr tty ts)
(close-port tty)
```

To show how to access fields in `struct termios` instances from Scheme, let's set the `ECHO` bit in the `c-lflag` field (remember that constants are named `termios-*` in `(termios system)`, which means that `ECHO` from the C library header file is available as `termios-ECHO`):

```
(define tty "/dev/ttyUSB0")
(define prt (open-io-file tty))
(define ts (make-termios-struct)
(tc-get-attr! tty ts)
(let* ((ts-scm (parse-termios-struct ts))
       (lflag (get-field-from-termios ts-scm 'c-lflag))
       (new-value (logior lflag termios-ECHO)))
  (put-field-into-termios! ts-scm 'c-lflag new-value)
  (tc-set-attr tty (make-termios-struct new-value)))
```

Finally, here is an example of how to use `call-with-errno` to properly access the value of `errno` by the use of `call-with-blocked-asyncs`:

```
(define tty "/dev/ttyUSB0")
(define prt (open-io-file tty))
(define ts (make-termios-struct)
(call-with-errno (errno (tc-get-attr! port ts))
  (strerror errno)
  (close port)
  (quit EXIT_FAILURE))
```

6 SEE ALSO

termios(3), *termios.h(7)*, *errno(3)*, *errno.h(7)*, *guile(1)* and the *Guile Reference Manual*

7 COPYRIGHT