Steve Friedl's Unixwiz.net Tech Tips

Understanding UNIX termios VMIN and VTIME

The POSIX "termios" structures are at the center of serial-port I/O control, and there are *many* knobs and switches to turn here. The **stty** program is actually a command-line wrapper around the **termios** struct, and it should be apparent that this whole arena is filled with arcana, obscura, historical artifacts, and even nostalgia.

A single Tech Tip can't possible cover them all, but it can at least touch on one area of common confusion: the use of **VMIN** and **VTIME**. These are macros used as indexes into the **termios.c_cc[]** array, which under normal circumstances holds the list of special *control characters* (backspace, erase, line kill, interrupt, etc.) for the current session.

But when the **ICANON** bit is turned off, a "raw mode" is selected which changes the interpretation of these values. These are used to guide the line-driver code in its decision on allowing the **read()** system call to return. We'll try to explain them in some detail.

> NOTE: This is **not** a tutorial on **termios** programming as a whole. This is a very large subject, and the only way to present a Tech Tip on a subject is to presume that the reader understands the subject in general.

An excellent resource is the dated but still timely book POSIX Programmer's Guide by Donald Lewine (O'Reilly & Associates). This covers terminal I/O quite well, along with other important POSIX topics (process control, signals, etc.). Highly recommended.

Who cares about timing?

Unlike reading from a file, where data are either "present" or "not present", there are also *timing issues* involved in reading from a tty channel. Many programs won't care about this at all, but there are some important cases where this is vital to correct operation.

Function-key processing

On a regular keyboard, most keys send just one byte each, but almost all keyboards have special keys that send a *sequence of characters* at a time. Examples (from an ANSI keyboard)

- **ESC [A** up arrow
- **ESC [5 ~** page up
- **ESC [18 ~** F7

and so on.

From a strictly "string recognition" point of view, it's easy enough to translate "**ESC [A**" into "up arrow" inside a program, but how does it tell the difference between "user typed up-arrow" and "user typed the ESCAPE key"? The difference is *timing*. If the ESCAPE is *immediately* followed by the rest of the expected sequence, then it's a function key: otherwise it's just a plain ESCAPE.

Efficient highspeed input

When a communications program (such as fax software) is reading from a modem, the data stream is arriving at a relatively high rate. Doing single-character-at-a-time input would be extremely inefficient, given that each **read()** involves a system call and an operating system context switch. We'd instead like to read in larger chunk if it's available for us, but still know how to recognize timeouts (which usually indicate error conditions).

Capturing occasional, low-volume data

When writing software that monitored a temperature sensor via a serial line, we expected to receive a short (10-20 bytes) message every second. This message arrived as a single burst, and once the first character of the message was received, we *knew* that the others were right behind it and would be completely received in about 100 milliseconds.

The message format did not have strong delimiters, so if we used a naïve read of so many bytes, we'd run the risk of reading an entire message but blocking until a few more bytes *of the next message* were read to fill our request. This could lead to getting "out of sync" with the sensor.

By setting **VMIN**/**VTIME** properly, we were able to insure that that we could efficiently capture all the data sent in one burst without risk of inter-message overlap.

We'll note that some of these timeout issues can be partly addressed by the use of signals and alarms, but this is really a substandard solution: signals and I/O are hard to get right (especially in a portable manner), and we very strongly prefer

using the features of the line-discipline code as they were intended. Signals suck.

When does read() return?

The **termios** settings are actually handled in the kernel, and the ones we're interested in are in the *line discipline* code. This sits above the "device driver", and this is consistent with our applying termios to both serial and network I/O (which obviously use different underlying hardware).

All of the **VMIN** and **VTIME** areas involve one question:

> *When does the line driver allow **read()** to return?*

With a "regular file", the operating system returns from the **read()** system call when either the user buffer is full, or when the end of file has been reached - this decision is easy and automatic.

But when the driver is reading from a terminal line, the "Are we done?" question can be asked over and over for each character that arrives. This question is resolved by setting **VMIN**/**VTIME**.

Using our temperature-sensor example, we'll list the requirements that inform our design:

- We normally read up to 20 bytes as a "message"
- Individual messages have their bytes all sent together
- No background processing required; while waiting for input, we're happy to block indefinitely waiting for something to happen

The last requirement means that we have no *overall* timeout, but we do have an *intercharacter* timeout. This is the key functionality provided by the line driver. Let's be specific.

When waiting for input, the driver returns when:

VTIME tenths of a second elapses *between* bytes

An internal timer is started when a character arrives, and it counts up in tenths of a second units. It's reset whenever new data arrives, so rapidly-arriving data never gives the intercharacter timer a chance to count very high.

It's only after the *last* character of a burst — when the line is quiet — that the timer really gets counting. When it reaches **VTIME** tenths, the user's request has been satisfied and the **read()** returns.

This provides exactly the behavior we want when dealing with bursty data: collect data while it's arriving rapidly, but when it calms down, give us what you got.

VMIN characters have been received, with no more data available

At first this appears duplicative to the **nbytes** parameter to the **read()** system call, but it's not quite the same thing.

The serial driver maintains an input queue of data received but not transferred to the user — clearly data can arrive even when we're not asking for it — and this is always first copied to the user's buffer on a **read()** without having to wait for anything.

But if we do end up blocking for I/O (because the kernel's input queue is empty), then **VMIN** kicks in: When that many bytes have been received, the **read()** request returns that data. In this respect we can think of the **nbytes** parameter as being the amount of data we *hope* to get, but we'll settle for **VMIN**.

The user's requested number of bytes has been satisfied

This rule trumps all the others: there is no circumstance where the system will provide *more* data than was actually asked for by the user. If the user asks for (say) ten bytes in the **read()** system call, and that much data is already waiting in the kernel's input queue, then it's returned to the caller immediately and without having **VMIN** and **VTIME** participate in any way.

These are certainly confusing to one who is new to **termios**, but it's not really poorly defined Instead, they solve problems that are not obvious to the newcomer. It's only when one is actually dealing with terminal I/O and running into issues of either performance or timing that one really must dig in.

VMIN and VTIME defined

VMIN is a character count ranging from 0 to 255 characters, and **VTIME** is time measured in 0.1 second intervals, (0 to 25.5 seconds). The value of "zero" is special to both of these parameters, and this suggests four combinations that we'll discuss below. In every case, the question is when a **read()** system call is satisfied, and this is our prototype call:

```
int n = read(fd, buffer, nbytes);
```

Keep in mind that the tty driver maintains an input queue of bytes already read from the serial line and not passed to the user, so not every **read()** call waits for actual I/O - the read may very well be satisfied directly from the input queue.

- **VMIN = 0 and VTIME = 0**

  This is a completely non-blocking read - the call is satisfied immediately directly from the driver's input queue. If data are available, it's transferred to the caller's buffer up to nbytes and returned. Otherwise zero is immediately returned to indicate "no data". We'll note that this is "polling" of the serial port, and it's almost always a bad idea. If done repeatedly, it can consume enormous amounts of processor time and is highly inefficient. Don't use this mode unless you really, really know what you're doing.

- **VMIN = 0 and VTIME > 0**

  This is a pure timed read. If data are available in the input queue, it's transferred to the caller's buffer up to a maximum of nbytes, and returned immediately to the caller. Otherwise the driver blocks until data arrives, or when VTIME tenths expire from the start of the call. If the timer expires without data, zero is returned. A single byte is sufficient to satisfy this read call, but if more is available in the input queue, it's returned to the caller. Note that this is an *overall* timer, not an *intercharacter* one.

- **VMIN > 0 and VTIME > 0**

  A **read()** is satisfied when either VMIN characters have been transferred to the caller's buffer, or when VTIME tenths expire between characters. Since this timer is not started until the first character arrives, this call can block indefinitely if the serial line is idle. This is the most common mode of operation, and we consider VTIME to be an *intercharacter* timeout, not an *overall* one. This call should never return zero bytes read.

- **VMIN > 0 and VTIME = 0**

  This is a counted read that is satisfied only when at least VMIN characters have been transferred to the caller's buffer - there is no timing component involved. This read can be satisfied from the driver's input queue (where the call could return immediately), or by waiting for new data to arrive: in this respect the call could block indefinitely. We believe that it's undefined behavior if nbytes is less then VMIN.

Home | Stephen J. Friedl | Software Consultant | Orange County, CA USA | steve@unixwiz.net |