Open Computing ‘‘Hands-On’’: ‘‘Wizard’s Grabbag’’ Column: February 94

Serial Data Captured

Need to record data being sent over an asynchronous serial line? Here’s a program that lets you store a data transmission to a file

Several years ago Frank Lees contributed a program useful for capturing characters sent to a Unix system over an asynchronous serial line. I decided to update the program to add POSIX support and a time-out feature designed with help from Ray Swartz. With this program, you can upload files to a Unix system using error-correcting modems and any terminal emulation program that can read or transmit files.

Dear Editor:

I use the upload C program [see Part A of the Listing] on my Unix system to capture files sent from PCs, intelligent terminals, or data-recording devices. It reads standard input and writes to a file, terminating when there is no input for a specified time, or 10 seconds by default. The Unix system asynchronous serial device driver is placed in ‘‘raw’’ mode so all characters can be collected without change. By default, Control-M and Control-Z characters are not stored in the output file but can be stored if so desired.

To upload a file using a communication program from another system, first invoke upload on the destination machine so it will begin storing characters that it reads from the serial port that is used for communication with the source system. Next request that the source system send the contents of the desired file through the communication port to the destination system. The upload program will ‘‘time out’’ after all the data has been transferred, finally closing the output file and restoring the original serial driver modes before exiting.

You must specify an argument on the `upload`-command line that names the output file. If this file exists, `upload` displays an error message and terminates. This feature prevents accidental overwriting of existing data.

Several options can be used to modify the default actions. Specify `-b` (for binary) to store all eight bits of every character received; otherwise, the most significant bit is stripped. Use `-m` to store Control-M’s and `-z` to store Control-Z’s if you’re not using binary mode. (DOS-based systems terminate lines with Control-M’s and end files with Control-Z’s.)

Normally, if no characters are received for 10 seconds, `upload` will close the output file and terminate. You can change this time-out period using the `-q` (for quit-time) option. Values will be rounded up to the next multiple of 10 (11 through 20 become 20, 21 through 30 become 30, and so forth). Also, values less than 10 seconds aren’t allowed because the basic timing loop requires at least 10 seconds to complete.

This program performs no error detection or correction. You'll have to rely on underlying protocols to provide error-free transmission. I developed `upload` on a Tektronix workstation running a BSD 4.2-like operating system.

Frank Lees / Woodbridge, N.J.

Usage Notes. Ensure that your Unix system input buffer doesn't overflow. The `upload` program sets the device driver on the destination system so it won't recognize XON (Control-Q) and XOFF (Control-S) characters, but the driver can still send XON and XOFF characters to the source system. The source system or communication program should recognize these characters and suspend transmission when requested by the destination machine to prevent loss of data.

You'll want to use binary mode to store all eight bits of every character received, which is necessary for compiled programs or 8-bit data. If the sending system includes parity bits when they send 7-bit data, they will be stored as the most significant bit in the output file. If you want to upload all characters of a text file, including Control-M's and Control-Z's, specify the `-m` and `-z` options with text mode (not binary mode) so any parity bits will be stripped.

Some of you may wonder: Why bother with `upload` when `cat -u >` *`output-file`* could do the job? Simply put, `upload` provides more options with only one command invocation. Otherwise, you'll need to use `stty` to set and then reset the device-driver modes and provide some sort of time-out facility.

Implementation Notes. As usual the program starts with constant definitions and variable declarations. However, provision has been made for portability between two major ways to control serial drivers--the POSIX ‘‘termios’’ structure and the System V ‘‘termio’’ structure-by using ‘‘pound define’’ constructs to select between two groups of definitions, declarations, and header-file ‘‘include’’ directives. Lines 2-5 will be compiled if `TERMIO` (for ‘‘termio’’ structure) is defined on the command line [see Part B of the Listing] or in a Makefile. Otherwise, we assume a POSIX environment, where lines 7-13 are employed. Note that many Unix versions besides System V support the ‘‘termio’’ structure, such as recent BSD and SunOS releases.

Line 7 defines `_POSIX_SOURCE` so any header files that are included subsequently will be interpreted with this constant in effect. This definition ensures that vendor extensions won't be used because only those symbols defined by the base standard or enabled by a specific #define feature test will be employed.

Line 79 rounds up the user-specified quit-time value to the next highest multiple of 10. On line 98 we open the output file so that if the file already exists, an error is returned. This approach prevents overwriting of data on the destination system disk. Part C shows one way to change the `open()` call so an existing file doesn't cause an error condition.

POSIX provides a portable approach to get and set the modes of an asynchronous serial driver. The implementation-specific `ioctl()` call has been replaced by serial-driver-specific subroutines because (1) you can't declare `ioctl()` with a standard C function prototype because the third argument varies in size and type depending on the second argument, (2) the calling semantics are different on different systems, and (3) international environments aren't supported adequately.

Serial-driver parameters are manipulated through a data structure. POSIX uses a structure named ‘‘termios’’ [Part D of the Listing], which is virtually identical to its predecessor, ‘‘termio’’ [Part E]. The primary difference is that ‘‘termios’’ doesn’t include the ‘‘termio’’ line-discipline member, `c_line`, which wasn’t used anyway. Also, many data types were made generic for portability. For instance, `tcflag_t` is specified as the data type for all the flag members instead of the more common unsigned short or unsigned long integer data types, and `cc_t` is used instead of an unsigned character.

Two nested loops do the work. The outer ‘‘while’’loop (Part A, lines 141-154) determines how long the program idles after the last character has been received. Each iteration of that loop decrements a loop count- which was originally set by line 140-to one-tenth the quit-time value. When this loop exits, the output file is closed, the serial-line I/O modes reset to their original values, and the program terminates after an end-of-transfer message is displayed for the operator.

Each iteration of the inner loop [lines 142-153] reads a character and then tests whether it’s to be written to the output file. Lines 146-147 perform several tests separated by the logical OR (`||`) operator. If any of these tests is true, the overall logical expression of the ‘‘if’’ statement is true, so the character will be saved. Otherwise, the character won’t appear in the output file.

If no input character is available, the ‘‘read’’ call [line 142] won’t be satisfied (will return control to the calling program) until the basic time-out period (10 seconds) has expired. This circumstance arises by using ‘‘noncanonical mode’’ (by disabling the `ICANON` structure flag). The time period is determined by the `VTIME` element of the `c_cc[]` serial-driver structure member. Multiply this value by 10 to get the period in seconds. Line 127 sets this value to 100 for a 10-second period. The Table lists all the ‘‘termio’’ or ‘‘termios’’ structure flags manipulated by this program.

Every time a character is read, the outer loop-count value is reset by line 152. This way, after the last character is read, the total quit time will have the correct value. This operation doesn’t seem to slow down the loop enough to effect data transmission, even at 9,600 bits per second.

Wanted: Testers

We’re looking for readers who have the inclination, time, and resources to help us test ‘‘Wizard’s Grabbag’’ contributions. This way we can provide a more portable program for our readers. In particular, interested readers who have access to one or more of the following Unix implementations: SCO Xenix 2.3.2, SCO Unix 3.2.x, DECstation Ultrix 4.1, or HP 700 HP-UX 8.07 should contact me via e-mail.

Edited by Becca Thomas / Online Editor / UnixWorld Online / beccat@wcmh.com

Go To Content | Search

Last Modified: Tuesday, 22-Aug-95 15:46:32 PDT