Linux Backspace/Delete mini−HOWTO

Sebastiano Vigna

vigna@acm.org

Revision History

Revision v1.3	18 October 2000
Name change.	
Revision v1.2	15 October 2000
Updated. Added "What If Nothing Works" section.	
Revision v1.1	13 September 2000
Added tcsh fixes	
Revision v1.0	5 September 2000
First release	

Table of Contents

1. Introduction

Every Linux user has been sooner or later trapped in a situation in which having working **Backspace** and **Delete** keys on the console and on X seemed impossible. This paper explains why this happens and suggests solutions. The notions given here are essentially distribution−independent: due to the widely different content of system configuration files in each distribution, I will try to give the reader enough knowledge to think up his/her own fixes, if necessary.

I assume that the **Backspace** key should go back one character and then erase the character under the cursor. On the other hand, the **Delete** key should delete the character under the cursor, without moving it. If you think that the function of the two keys should be exchanged, in spite of the fact that most keyboards feature an arrow pointing to the *left* ( ) on the **Backspace** key, then this paper will not give you immediate solutions, but certainly you may find the explanations given here useful.

Another assumption is that the fixes should alter only local (user) files. No standard part of the distribution should be altered. Finally, this document discusses how to set up your system so that applications get the right events. If an application decides to interpret such events in an idiosyncratic way, the only possible fix is to reconfigure the application.

2. How Keys Are Turned Into Actions

When a key is pressed on the keyboard, a number of hardware and software components cooperate so as to guarantee that the intended meaning of the key (e.g., emitting a certain character) matches the actual behaviour of the key. I will concentrate on the software side (as our control on the hardware part is nonexistent), and in particular, for the time being, on the events related to console output.

1. Hitting a key causes raw keyboard *scancodes* to be generated; these scancodes are then transformed in a *keycode*. On an i386 system, usually the key **Backspace** emits 14 and the key **Delete** emits 111.
2. The keycodes are translated by the keyboard library into a *keyboard symbol (keysym)* using the keyboard definition loaded by the user. If you look into your keyboard database (e.g., under Red Hat Linux, in `/usr/lib/kbd/`), you'll discover several definitions for different computers, different layouts and possibly different interpretations of the same keys (e.g., one could desire that the two **Alt** keys really behave as distinct modifiers). The Linux console keyboard layout assigns keysym Delete to keycode 14 and keysym Remove to keycode 111. This may seem strange, but the Linux console emulates a VT102 terminal, and this is the way things work in that realm.
3. Our journey has still to come to an end. Console applications read ASCII sequences, not keysyms. So the console must read keysyms and translate them into ASCII sequences that suitably encode the keys. Of course this operation must be performed in a way that is understandable by applications. For instance, on the Linux console the Delete keysym is mapped to the ASCII code 127 (DEL), the Remove keysym on a suitable escape sequence, and the BackSpace keysym to ASCII code 8 (BS).
4. Finally, we must in a sense roll back to what we had before and translate the ASCII sequences generated by each key into a *key capability*. This goal is reached by a *terminal database*, which contains, for each kind of terminal, a reverse mapping from sequences of characters to key capabilities (which are essentially a subset of the keysyms).

   Note: Unfortunately, there are two "standard" terminal databases, termcap and terminfo. Depending on your distribution, you could be using either one of them, or the database could even depend on the application. Our discussion will concentrate on the more modern terminfo database, but the suggested fixes take both into consideration.

For instance, on the Linux console **F1** generates an escape followed by `[[A`, which can be translated to the capability `key_f1` by looking into the terminal−database entry of the console (try **`infocmp linux`** if you want to have a look at the entry). A very good and thorough discussion of terminal databases can be found in GNU's termcap manual. Usually Linux applications use the newer terminfo database, contained in the ncurses package.

Maybe at this point not surprisingly, the Linux console terminfo entry maps DEL to the `kbs` (backspace key) capability, and escape followed by `[3~` to the `kdch1` ("delete−one−char" key) capability. Even if you could find strange that the **Backspace** key emits a DEL, the terminal database puts everything back into its right place, and correctly behaving applications will interpret DEL as the capability `kbs`, thus deleting the character to the left of the cursor.

3. Why It Doesn't (Always) Work

I hope the basic problem is clear at this point: there is a bottleneck between the keyboard and the applications, that is, the fact that they can only communicate by ASCII sequences. So special keys must be first translated from keysyms to sequences, and then from sequences to key capabilities. Since different consoles have different ideas about what this translation can look like, we need a terminal database. The system would work flawlessly, except for a small problem: not everyone uses it.

More precisely, many application *do not use* the terminal database (or at least not all of it), and consider BS and DEL ASCII codes with an intended meaning: thus, without looking at the database, they assign them semantics (usually, of course, the semantics is removing the character before or under the cursor). So now our beautiful scheme is completely broken (as every Linux user is bitterly aware). For instance, the bash assumes that DEL should do a backward−delete−char, that is, backspace. Hence, on a fresh install the **Backspace** key works on the console as expected, but just because of two twists in a row! Of course, the **Delete** key does not work. This happens because the bash does not look into the terminal database for the `kdch1` capability.

Just to illustrate how things have become entangled, consider the **fix_bs_and_del** script provided with the Red Hat distribution (and maybe others). It assigns on−the−fly the BackSpace keysym to the **Backspace** key, and the Delete keysym to the **Delete** key. Now the shell works! Unfortunately, all programs relying on the correct coupling of keysym generation and terminal database mappings are now not working at all, as the Delete keysym is mapped to DEL, and the latter to the `kbs` key capability by the terminfo database, so in such programs both keys produce backspacing.

4. X

The situation under X is not really different. There is just a different layer, that is, the X window system translates the scancodes into its own keysyms, which are much more varied and precise than the console ones, and feeds them into applications (by the way, this is the reason why XEmacs is not plagued by the problem: X translates keycode 14 to keysym BackSpace and keycode 111 to keysym Delete, and then the user can easily assign to those keysyms the desired behaviour). Of course, a terminal emulator program (usually a VT100 emulator in the X world) must translate the X keysyms into ASCII sequences, so we are again in our sore business.

More in detail, xterm behaves exactly like the console (i.e., it emits the same ASCII sequences), but, for instance, gnome−terminal emits BS for **Backspace** and DEL for **Delete**. The real fun starts when you realise that by default they use the *same* terminal−database entry, so the fact that the `kbs` capability is associated to an ASCII DEL makes all correctly behaving applications produce the same behaviour for the **Backspace** and **Delete** keys in gnome−terminal. The simple statement

```
bash$ export TERM=gnome
```

can solve the problem in this case for correctly behaving applications. Well, not always, because your system could lack an entry in the terminal database named `gnome`, in particular if it is not particularly up−to−date.

5. What You Should Do When Writing Applications

When you write a console application, be kind to the user and try to understand what comes from the standard input using the following fallback chain:

1. open the right terminfo entry, and try to process the sequence so as to discover whether it has a particular meaning on the current terminal; if so, use the terminfo semantics;
2. use the ASCII intended meaning on line feeds, newlines, tab characters and, of course, BS and DEL. Crossing your finger could also be useful.

6. What You Should Do On Your System

Note again that the main issue that confuses people trying to fix their system is that usually they are fixing thing in the wrong place. Since the parts that work often just work by chance, trying to fix the system assuming something is broken will usually lead to change correct settings into incorrect settings.

6.1. What Needs to Be Done

6.1.1. Distinguishing Between Emulators

The first step towards a clean solution is to get rid of the confusion between gnome−terminal and xterm, and, more generally, between different terminal emulators. Theoretically, the option `termname` should allow the user to set the `TERM` variable to a more sensible name, for instance `gnome`. However, as of gnome−terminal 1.2.1 the option does not work.

The idea here is to exploit the fact that gnome−terminal sets the `COLORTERM` variable to `gnome−terminal`. Thus, by adding a simple test to the shell configuration files we can fix the `TERM` variable.

6.1.2. Fixing the Terminal Database

Our problem now is that the terminal database could lack an entry named `gnome` (this happens on a number of termcap and terminfo versions). Recent terminfo databases have an entry, but since gnome−terminal essentially emulates xterm modulo a couple of keys, it is possible to automagically generate a brand new correct entry. This will fix the behaviour of all correct applications using the terminal database.

6.1.3. Fixing the Shell Behaviour

The readline library used by the bash and by many other programs to read the input line can be customized so to recognize specific sequences of characters. The customization can also depend on the `TERM` variable, so once we can distinguish terminals we can do fine tuning of the keyboard.

Moreover, if you want less and other application that do raw line input to work correctly, you must convince the shell that under gnome−terminal the erase character is BS, and not DEL (in the other case the **Backspace** key is already emitting DEL, so we do not have to do anything). This can be done using the command **stty**.

6.2. How to Do It

Caution

These fixes have some drawbacks. First, they work only for the specified terminals. Second, in theory (but this is unlikely to happen) they could confuse the readline library on other terminals. Both limitations are however mostly harmless.

First of all, check with **`infocmp gnome`** whether you already have a `gnome` entry in your terminfo (we will fix termcap later). If the entry does not exist, the following command

```
bash$ tic <(infocmp xterm |\
        sed 's/xterm|/gnome|/' |\
        sed 's/kbs=\\177,/kbs=^H,/' |\
        sed 's/kdch1=\\E\[3~,/kdch1=\\177,/')
```

will create a correct one in `~/.terminfo`. If the same command is launched by the root, it will generate the entry in the global database (you can override this behaviour by setting `TERMINFO` to `~/.terminfo`).

Now, add add the following snippet to `~/.inputrc`[1]:

```
"\e[3~": delete-char

$if term=gnome
DEL: delete-char
$endif
```

The first line makes the **Delete** key work on the console and on xterm, and with a bit of luck it should not interfere with other terminals. The other conditional assignment makes gnome−terminal work *given that the `TERM` variable is set correctly*. To guarantee this, add

```
if [ "$COLORTERM" = "gnome-terminal" ]
then
    export TERM=gnome
fi

if [ "$TERM" = "gnome" ]
then
    export TERMCAP=$(infocmp -C gnome | grep -v '^#' | \
                    tr '\n\t' '  ' | sed 's/\\  //g' | sed s/::/:/g)
    stty erase ^H
fi
```

to your `~/.bashrc` configuration file[2]. The assignments are executed only under gnome−terminal: the first **export** sets correctly the `TERM` variable, the second one creates a local termcap entry deriving it from the terminfo one[3], and finally we use **stty** to instruct the shell that the erase character is really BS.

> **Note:** If you have a Red Hat older than 6.0, then you must add also
>
> ```
> $if term=xterm
> DEL: delete-char
> $endif
> ```
>
> to your `~/.inputrc` file and
>
> ```
> if [ "$TERM" = "xterm" ]
> then
>     stty erase ^H
> fi
> ```
>
> to your `~/.bashrc` file. The keyboard policy of Red Hat changed with 6.0 in a

backward−incompatible way, so there is nothing you can do about this.

Note: Later versions or different distribution could have fixes already in place in the system−wide `/etc/inputrc` configuration file, as it happens, for instance, in Red Hat 7.0. In this case you can eliminate redundant lines from your `~/.inputrc`. Moreover, later versions of gnome−terminal could implement the `termname` option correctly. In this case, a cleaner way of getting the `TERM` variable set correctly is to invoke gnome−terminal with **gnome−terminal −−termname=gnome** (and you can eliminate the check on the `COLORTERM` variable).

6.3. Fixing for tcsh

In the case of the tcsh, the fixes go all in `~/.tcshrc`:

```
bindkey "^[[3~" delete-char

if ($?COLORTERM) then
   if ($COLORTERM == "gnome-terminal") then
      setenv TERM gnome
   endif
endif

if ($?TERM) then
   if ($TERM == "gnome") then
      setenv TERMCAP \
       "`infocmp -C gnome | grep -v '^#' | tr '\n\t' '  ' | sed 's/\\  //g' | sed s/::/:/g`"
      bindkey "\177" delete-char
      stty erase ^H
   endif
endif
```

Note: Of course, if you have a Red Hat older than 6.0, then you must add also

```
if ($?TERM) then
   if ($TERM == "xterm") then
      bindkey "\177" delete-char
      stty erase ^H
   endif
endif
```

to your `~/.tcshrc` file.

7. What If Nothing Works

The first thing to do is understanding which ASCII codes are produced by a certain key. The following C one−liner

```
void main(void) {int c; while(c = getchar()) printf("%d %02X\n", c, c);}
```

may help you. Put the line into a file named `ascii.c`, compile it with **gcc ascii.c −o ascii**, type **./ascii** and press a key followed by **RETURN**. The program will display the decimal and hexadecimal codes of the ASCII sequence produced (you may want to do a **stty erase** `^−` first to get really all the codes).

Once you know which sequences are produced, you must check the current terminfo entry with **infocmp** (don't be scared by the amount of information printed!) and be sure that the `kbs` and `kdch1` capabilities correspond to the right sequences (that is, the one produced by the respective keys).

If there is a mismatch, there can be several different reason: wrong content of the `TERM` variable, wrong entry of the terminal database, wrong terminal emulation under X. I hope at this point you have enough information to dig the solution autonomously.

> **Note:** If different applications behave in different ways, it is likely that some of them are using the terminal database correctly, and some are not. Remember that the fact that the keys produce the right behaviour in a certain application does not mean that the application is using correctly the terminal database they could work just by chance. If you want to have an independent check, you can try whether the **ne** editor works. **ne** uses all terminal capabilities, including `kbs` and `kdch1`, and uses intended meaning only as a last resource.

8. Conclusions

The fixes suggested here should solve to a large extent the problem of deleting text you wrote (however, they do not help in creating other text `:)`).

There is a small bug in the whole setting: if you start xterm from gnome−terminal, it will get `TERM` set to `gnome`. This inconvenience is, of course, harmless, and it will be solved as soon as it will be possible to start gnome−terminal with `TERM` suitably set.

Finally, it should be noted that the fixes will not work for broken applications (for instance, applications ignoring the `kbs` key capability). There is little to do in this case, as fixing for one broken application will likely break all well−behaving ones.

Notes

[1]

On older version of the bash, you must remember to set `INPUTRC` suitably, for instance adding

```
export INPUTRC=~/.inputrc
```

to your `~/.profile` (or whichever file is read just by login shells).

[2]

More precisely, to the shell configuration file that is read in every shell, not only in login shells. The right file depend on startup sequence of your bash.

[3]

However, as terminfo entries gets bigger and bigger, this method could generate too large a termcap entry.