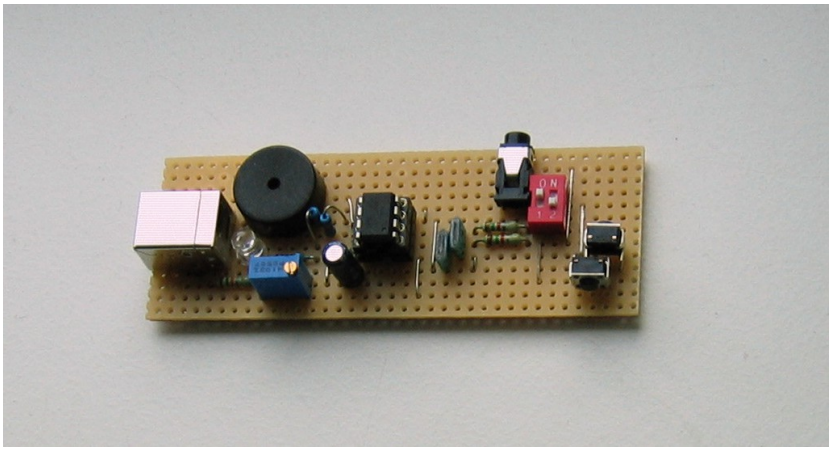
VUSB Morse Keyboard

Ralf Beesner

March 22, 2014



1 Summary

There are some folks, smarter than me, who developed two USB software libraries that empower a simple, cheap AtTiny or AtMega micro controller to work as a USB device.

Of course, Atmel sells some bigger AVR controllers with a hardware USB interface, but using the software libraries, there is no need to fiddle around with 64pin-SMD devices. Many projects use a simple AtTiny 45 in an 8-pin DIP package, a 20-pin AtTiny 2313 or an AtMega 8 in a 28-pin DIL-package.

In recent years, I built some USB gadgets (developed by other people also smarter than me) and made only small changes in the source code to make the devices better fit my preferences.

Some of the projects emulate USB keyboards, which I find interesting, because you need no driver and no application software on the host PC (the devices are just recognized automatically).

The emulated keyboard may have special keys (that are not available on a standard keyboard) or serve a limited function, e.g. a slide show presenter with only two keys for "forward" and "backward".

The latest projects I built are two morse telegraphy keyboards, and I had to add some bigger chunks of own code.

So, whats a morse keyboard? It is a virtual USB keyboard with only one key (or two keys), which translates an operator's morse telegraphy input into keyboard key presses. If you know morse code, you may use it to write into a PC application (e.g. mailer, word processor or spread sheet program) by using a morse keyer. There are two flavors: the first one serves a straight morse key (a straight key works more or less like a doorbell switch), the second emulates an electronic morse keyer (elbug), which is operated using a paddle switch or (two) squeeze paddle switches.

2 Software

There are two AVR software USB libraries available, one is the VUSB library `www.obdev.at/products/vusb/index.html`, the other is the BASCOM-SWUSB-Library `www.sloservers.com/swusb` .

As I am a lousy programmer, I preferred BASCOM `www.mcselec.com`. The Basic dialect is much easier than the C language, but has it's limits. One of the limits being the SWUSB-library, which is closed source (its core is a binary named swusb.lbx) and doesn't seem to be maintained any more by its creator.

So my BASCOM morse keyboard `www.elektronik-labor.de/AVR/USBbascomCW.html` didn't make me happy any more.

VUSB, however, is licensed under the GPL and seems to be maintained periodically by its creator.

Imho C is a rather horrible language for micro controllers (for example, it has no commands for direct bit manipulations, so you have to use cryptic workarounds), but the VUSB library is so useful that I tried to write two C programs. I wrote only the morse routines, most of the keyboard code is "borrowed" from the 4-Key-Keyboard (Author: Flip van den Berg - `blog.flipwork.nl`).

My code may not be elegant, as I am a bloody beginner in C, but both versions are working reliably and fit easily into an 8-pin AtTiny45.

A morse decoder may be quite easy, if you use blocking code, e.g. delays, to decide if a morse element was a short one (dit) or a long one (dah), or if a pause was a space between morse elements or between characters.

But as the USB protocol is time-sensitive, the interrupts and routines of the USB library must have priority. So the morse part has to work without delays, and therefore it's a bit more complicated, it is a sort of state machine.

Timer0 generates the morse sidetone, Timer1 generates overflows every 4 ms, and the overflow flag is polled by the program. The program counts (increases) the number of overflows, and steps through the state machine. When a timing event has occurred (e.g. the length of a "dit" or "dah" was exceeded), the state machine jumps into the next state, until a morse character is completed.

The bit pattern of the morse character is looked up in a table. The table is provided by the include file "morsetable.h". It returns the ASCII value of the morse character. However, the HID (Human Interface Device) keyboard table is different from the ASCII table, so the ASCII byte has to be remapped into a HID byte. The mapping table is provided by the include file "hidkbd-??.h". The HID byte is then handed over to the VUSB routines.

Unfortunately, all HID keyboards behave like US keyboards. On a German keyboard, when you press the "Z" key, the keyboard hands a "Y" to the PC and it is the task of the PC keyboard driver to convert it into a "Z", which is then presented to the application and to the user.

So the morse keybord must send an "Y", too, when it wants to present a "Z" to the user. Therefore, the HID mapping table has to be modified, if you don't want to switch your PC's keyboard driver to "US English" temporarily.

There are two HID tables provided - the standard HID table hidkbd-us.h and a patched version named hidkbd-de.h. One of them is chosen by an include statement at the beginning of main.c .

If you want a different keyboard layout, you must try to patch it by yourself.

By the way, on a Linux machine you may switch the keyboard language temporarily by the command "setxkbmap us , "setxkbmap de" or for example "setxkbmap fr".

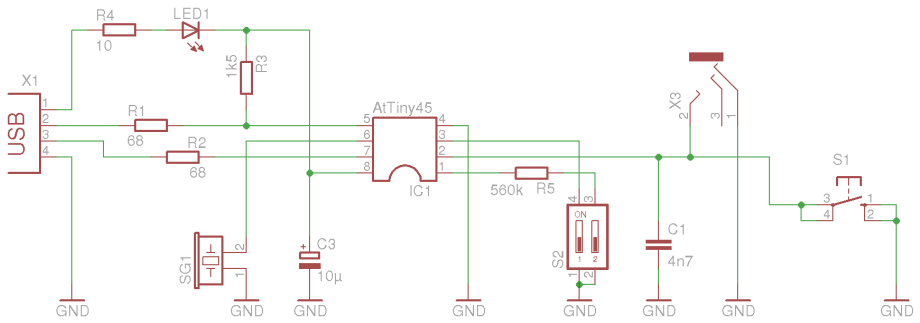


Figure 1: straight key morsekeyboard

3 Hardware

The electronic circuit is quite simple. The straight key version differs a little from the elbug version. The straight key version has one key and two switches. Switch 1 is used to choose the speed range - either from about 50 characters per minute (cpm) to 80 or 90 cpm, or from about 80 cpm to 120 cpm. Switch 2 activates the autoSpace function. As there was no free digital input pin available, I misused the analog input shared with the reset pin - as long as the voltage keeps above 50 %, the chip does not reset (this trick avoids fusing the reset pin away).

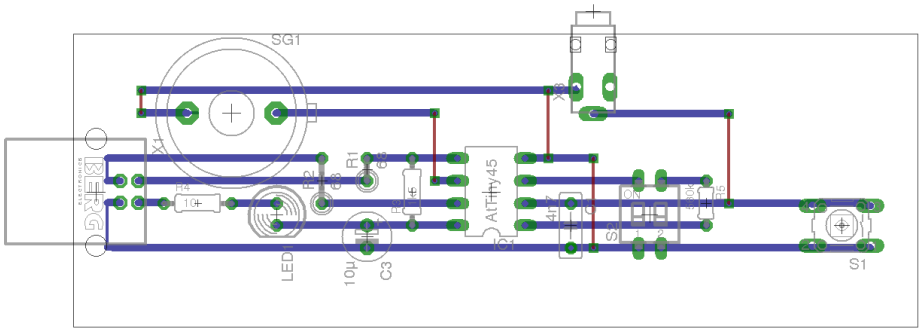


Figure 2: Layout straight key morse keyboard (seen from top side)

The elbug version uses the analog input shared with the reset pin, too. The 10 kOhm potentiometer decides the elbug morse speed, and the voltage is kept above 50% by a 15 kOhm series resistor, which limits the voltage swing of the potentiometer.

There are two choices for the paddle - you may use an external one plugged into the 3.5 mm stereo jack or the two vertical dip buttons, which are arranged as a tiny provisional squeeze keyer.

The morse keyer inputs are debounced by hardware to keep the software more simple. So. the capacitors C1 and C2 are important and may not be omitted!

The USB part follows the standard recommendation by www.obdev.at.

The photo of my prototype hardware is the elbug version, but it differs a little, because I added a dip switch to convert the elbug hardware into the straight key hardware.

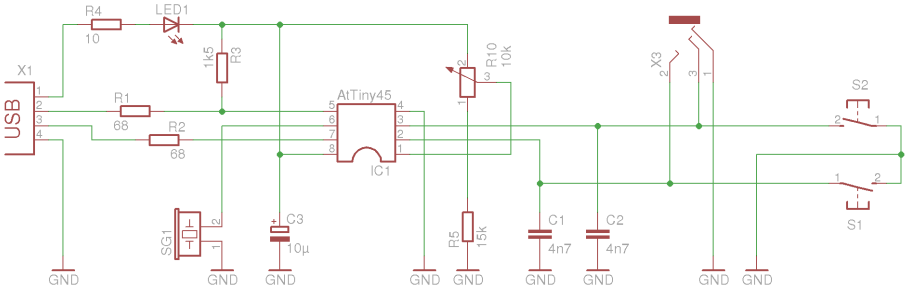


Figure 3: elbug morse keyboard

4 Compiling and Fusing

I didn't use the AVR Studio under Windows, but AVR-GCC under Linux. Anyway, AVR Studio uses WinAVR, which is the Windows version of AVR-

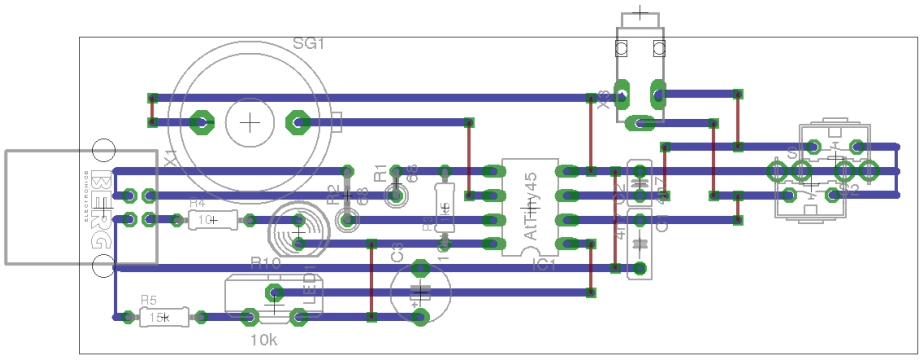


Figure 4: Layout elbug morse keyboard (seen from top side)

GCC. So compilation should be possible without the AVR Studio mumble by just opening a shell (called "DOS window" by some folks), changing to the folder with the source code and typing "make" and "make flash". I use an USBASP programmer, if you use a different one, you may edit the programmer section of the Makefile.

When the AtTiny45 is delivered from factory, it is clocked by its internal 8 MHz RC oscillator and its 1/8 prescaler. It must be re-fused to use its PLL clock (which is aligned to 16.5 MHz at runtime of the VUSB software). You may re-fuse it by the command "make fuses".

If you prefer a different program for changing the fuses, this are the correct bytes: HFUSE: 0xDD LFUSE: 0xE1 .

5 Usage

Using the elbug version, dits and dahs are formed by pressing the paddles, the right one produces dahs and the left one produces dits. They are automatically timed to the proper length. Pressing both paddles, produces a dit-dah-dit-dah-dit sequence. The Squeeze Mode should be Mode A, it may be changed to Mode B by deleting an if-condition in man.c and recompiling the code (hopefully, as I don't have much practice with squeeze keyers).

The morse speed is adjusted by the potentiometer.

As the morse code has no characters for <return> and <blank>, two morse traffic abbreviations are used:

Return: <kn>

Blank: <as>

I added a third one which is handy for jumping from left to right in an Excel or [Open|Libre]Office calculation sheet:

TAB: <ab>

The device has an autoSpace function (autoSpace inserts blanks automatically when the pause was long enough to indicate a space between words). As there was no free I/O-pin any more, autoSpace is activated by pressing the dit paddle while the device is plugged into the USB.

Using the straight key version, switch 1 is used for roughly choosing the morse speed, switch 2 is used to activate the autoSpace function.