Keyboard Translation for CP/M Plus running on yaze-ag

Jon Saxton
March 2010
Updated January 2015

This document describes the nature, implementation and rationale for a feature added to yaze-ag 2.3 which helps to make keyboard cursor keys usable within the yaze-ag emulator.

Making this work properly has been an adventure and I have learned far more about keyboards than I ever thought I would want to know. But it has been interesting and I have been able to enjoy the fruits of my own efforts in that screen navigation in CP/M Plus is now simple and intuitive.

There is no need to use key translation at all. If you are happy with navigating around CP/M and its applications using the standard PC keys (including the use of the Ctrl key) then you will not lose anything by ignoring key translation; yaze-ag will continue to work.

The problem

Andreas Gerlich's customised version of yaze (Yet Another Z80 Emulator) runs on UNIX and UNIX-like platforms which include Linux, Solaris and Macintosh. It even includes Windows systems running the Cygwin emulation layer. For better or worse in each of these environments the keyboard emulates that on a VT-100 terminal or something quite similar. The problem for CP/M programs is that many of the keys generate multi-byte sequences.

In general, CP/M and application programs running under CP/M respond to single-byte keypress events. Even word processing programs like WordStar deal with multi-byte sequences as a series of single-byte events and do not handle a sequence like

<escape> [1 9 ~

as something atomic (F7 in this case). Keys like ↑, Home and F7 are not likely to have useful effects in a CP/M application.

The ambition

The first objective was to make the cursor keys (↑, ↓, ←, →, Page Up, Page Down, Home and End) send something useful to CP/M.

A second objective was to intercept multi-byte sequences generated by other keys and prevent those sequences from being passed through to the CP/M environment.

During the early stages of writing the code for the first two goals a third emerged as a natural extension. That was to offer a keyboard macro facility where any key, perhaps combined with one or more modifiers (Shift, Control, Alt), could be programmed to send any sequence of characters to CP/M.

Observations

1. There are minor differences in the keyboard configuration between the various environments available for testing.

2. Most multi-byte sequences generated by cursor keys, function keys and Alt key chords begin with an ESC (escape) character.

3. Characters outside the ASCII set (e.g. £, ü, ß, ê, €) generate UTF-8 sequences.

4. Amongst CP/M applications, the most common keystrokes for controlling program execution and cursor movement is the set made popular by WordStar.

5. The CP/M command editing controls do not follow the WordStar pattern.

This last two points warrant elaboration.

There was no such thing as a standard for keystroke interpretation in CP/M. One text editor might interpret ^I (control-I) as an instruction to "move the cursor up one line" whereas another might respond by inserting some spaces into the text. Quite simply, there were no rules and nobody would have been obliged to follow them anyway. Notwithstanding the complete absence of documented standards, a *de facto* standard was established by MicroPro with its early word processing program, WordStar. For better or worse many other programs dealing with text in some form or other adopted the same or very similar conventions with respect to keystroke interpretation. While not universal, the WordStar key conventions are the ones which should be established as the default. Ideally, those conventions should be changeable at the whim of the user for personal preference or to accommodate software which uses different keys.

The CCP gets its command line input via BDOS function 10 and so the line editing functions are determined by the internals of the BDOS. Indeed any function which gets terminal input a line at a time works that way, not just the CCP. Yaze-ag uses Simeon Cran's ZPM3 as a CP/M+ BDOS replacement and even though the editing functions are much closer to the WordStar model than those of the standard Digital Research BDOS there are some troublesome incompatibilities. However there is a solution; it will be described later.

Implementation

The keyboard input routine in yaze-ag recognises single characters, escape-prefixed multi-byte sequences and UTF-8 sequences and constructs an internal representation of the key that was pressed. The internal representation is then handed to a translator module which checks to see if a translation has been defined for that key in which case the corresponding sequence of characters is queued for sending to CP/M. If no translation was specified then the keystroke is simply discarded.

Single-byte keystrokes are implicitly defined to have identity translations. In other words, it is not necessary to specify translations for things like P, <enter>, %, 6 and control-E; they will be passed to CP/M unaltered.

Of course you can define translations for single-byte keystrokes if you wish and such definitions will supersede the implicit ones thus allowing you to completely remap the keyboard for CP/M should you be so inclined.

Key translations are specified outside of CP/M in simple text files which can be prepared on the host platform with any ordinary text editor. The default translation file is yaze.ktt and that will be loaded if it exists. Within CP/M the KTT command can be used to switch to a different file, or to unload all translations. (When no translation is loaded, keystroke filtering is still in effect so multi-byte sequences will not be sent to CP/M.)

The idea of being able to switch translations is to cater for special applications which respond to unusual (dare we say "non-standard") keystrokes. Before running such a program tell yaze-ag to load an appropriate key translation table.

For example, consider the game of ladder. You have to configure ladder with some movement keys corresponding to the directions left, right, up and down. You cannot choose control keys so if you have a numeric keypad you'd probably pick 4, 6, 2 and 8 respectively. If you're using a laptop then you're probably stuck with some pattern amongst the letters but a more convenient option might be to write a key translation table.

```
left = "4"
right = "6"
up = "2"
down = "8"
```

A game of ladder could be run via a submit file:

```
ktt ladder          [Switch to ladder.ktt]
```

ladder	[Play game]
ktt yaze	[Switch back to normal operation]

Keyboard input is captured in "raw" mode but even so, yaze-ag does not get access to the lowest level of input. By the time that keystrokes have been delivered to yaze-ag they have already been processed by the host operating system in that scan codes have been translated into byte sequences. What this means is that some keystrokes never make it into yaze-ag because they are intercepted by the host system. For example, on ubuntu Linux, F1 invokes the "help" system and F11 flips the current (xterm) window between normal and maximised display. This just means that a few keys are unavailable for translation because yaze-ag never sees them.

Examples of key translation definitions

home = SOH	Home key sends ^A to CP/M.
F8 = "a6: ! dir"	Function key 8 delivers a command to log into drive A, user 6 and display the directory.
left = ^s	Left arrow key sends ^S to CP/M.
ctrl down = ^X	Control key plus the downward arrow key sends ^X to CP/M.
PageUp = 12	PgUp key sends ^R to CP/M
U+2F = bs	Pressing the / key sends a backspace to CP/M. (Don't ask why you might want to do that but you can.)
u+20ac = "$"	Translate the Euro symbol (€) to a dollar ($) for CP/M.

Writing a translate table

<u>File names</u>

As mentioned above, keyboard translations are specified in plain text files and you can have as many such tables as you want, loading them at will.

The translate table file names should be in lower case because the KTT program will pass a lower case file name to yaze-ag for loading. Also, it is convenient to have the file names end with ".ktt" because yaze-ag will look for that form of name if it doesn't find the one you specify. For example:

```
ktt kangaroo
```

tells yaze-ag to load "kangaroo" if it exists, otherwise to try "kangaroo.ktt".

(Note that because the files live in the host file system the names are not limited to an 8.3 pattern.)

Overall structure of a translation file

A key translation file comprises comments and key mappings. Nothing else. Each key mapping is independent and there is no required order.

Any line where the first non-whitespace character is not alphabetic or numeric will be treated as a comment and ignored.

Example:

```
; This is a comment
   –  and so is this
      but yaze-ag will try to make sense of this line
and this one too.
```

The interesting lines are the ones which are not comments. For one of those to be useful it should be of the form:

```
<key descriptor> = <substitute sequence>
```

A <key descriptor> tells yaze-ag to handle a specific key and <substitute sequence> tells yaze-ag what you want to send to CP/M when that key is pressed.

Some simple examples:

```
F6 = "789"
Up = ^W
```

Key descriptors

A key descriptor can be one of the following:

- A single character such as 3 or + or q
- A key name such as Home, Right, PageUp or F9
- A unicode code point such as U+1B or u+20ac
- A character name such as Escape or Equals (the only two)

Furthermore, the descriptor can be accompanied by one or more modifiers, specifically Shift, Control or Alt. There is also a fourth modifier, Literal, which does nothing. Its purpose is described later.

Examples:

```
y
Del
shift F10
control shift end
alt p
alt P
```

A bit of common sense is needed when using modifiers. The combination

```
shift r
```

is never seen by yaze-ag because pressing the shift key in conjunction with the r key yields R. In the examples listed above, note that the last one is generated by pressing the Alt, Shift and P keys but the shift modifier is not seen, only the Alt and P. Nonetheless, Alt-p and Alt-P are different.

A general rule is that a shift modifier is only visible when there is no glyph associated with a key. Another is that modifiers on a Unicode code point are meaningless (and are discarded.)

Suppose you wanted to re-map the grave accent key (to the left of the 1 key on the US keyboard) to something else. Now you can't put the grave accent as the first thing on the line because it is not a letter or a number so the line would be treated as a comment. There are two workarounds:

1. Specify the character in Unicode (e.g. U+60 for the grave)
2. Use the "Literal" modifier.

The keyword "Literal" (or "Lit") can be put at the start of a line to stop the line from being discarded as a comment, e.g.:

```
Literal \
lit `
```

As this example suggests, some of the keywords can be abbreviated:

ctrl	is equivalent to	control
lit	is equivalent to	literal
esc	is equivalent to	escape

and so on. A complete list of keywords and key names is given in the appendix.

<u>Substitute sequence</u>

Once a keystroke is recognised as having a translation, the stuff after the equals sign is what gets sent to CP/M. Very often that is just a single character but you can tell yaze-ag to send more. One example I used in testing was:

```
alt \ = etb stx eot "Boo!"
```

which did interesting things on the CP/M command line.

OK, so what are all the things following the = sign? The "Boo!" is pretty obvious. ETB, STX and EOT are character names. Another way to write them is ^W ^B ^D which is a fairly conventional notation for control characters and probably more familiar to CP/M users. Yet another way is to use two hexadecimal digits, 17 02 04 in this case. And of course you can mix them up - 17 stx ^D means the same thing.

Beyond using character names the general rule is that almost every character with a glyph can be enclosed in "quote" marks, any control character can be specified with a ^ prefix and any character at all can be specified as a pair of hexadecimal digits. The only exception to the quoted string is the quote itself; for that you have to use the hexadecimal notation (22) or a character name, e.g:

alt / = "This is a string with " 22 "quote" <u>quote</u> " marks"

This yields

```
This is a string with "quote" marks
```

when Alt-/ is pressed. See how the first quote mark is generated by the hex number 22 and the second is generated by the keyword which is underlined in the text above.

<u>Mapping the keys</u>

It is easy enough to write the translations, just decide what needs to be intercepted and what should be sent to CP/M. Finding out what needs to be intercepted means knowing what each interesting key generates when pressed. That information is used to code the keystroke recogniser state machine in yaze-ag itself (conin.c).

The keystroke processing path is as follows ...

- User presses a series of keys. Those keys generate a stream of characters, sometimes several characters per keypress.
- The key recognition state machine parses the input stream and identifies each key which was pressed and whether or not it was chorded with one of the shift keys, i.e. shift, control, alt or possibly some combination thereof.
- Each key identifier is passed through the translate module which uses the keyboard translate table (read in from an external file such as yaze.ktt) to see what, if anything, should be queued for CP/M.

- When CP/M requests keyboard input a byte is taken from the head of the queue unless the queue is empty in which case the whole system waits for input from the user.

The keystroke recogniser is fairly complete and has been tested on several UNIX-like environments including several 32- and 64-bit linux distributions, Solaris 10 and on Windows (accompanied by the appropriate 32- or 64-bit cygwin1.dll).

It should only be necessary to alter the state machine if yaze-ag is ported to a rather different environment or if some useful key is not recognised by its name. Should that need arise then there is a test program, appropriately named "keytest", which has two useful functions. The first is to report what codes are generated when you press a key. That information is critical to coding the recogniser. The second is to show what happens after a recognised key is passed through the translation module using a translate table, either the default yaze.ktt or a different one.

<u>Sample sessions</u>:

```
./keytest -
Press keys.  <esc> <esc> <esc> to end.
<esc>[24~<esc>[24;5~<0D>
<E2><82><AC><0D>
<esc><esc><esc>
```

In this session the keytest program is run in keycode reporting mode. The line starting with <esc>[24~ shows the sequence of bytes generated by F12 followed by ctrl-F12 and the Return key. The next line shows what the Euro key (AltGr-5 on my keyboard) generated, namely the UTF-8 sequence for €. The last line of three consecutive ESC keystrokes ended the program.

```
./keytest +
Press keys.  <esc> <esc> <esc> to end
<esc><esc>
key = 0000001B
<esc>
```

This session shows the result of running keytest in translate mode using the default translate table, yaze.ktt and pressing exactly the same keys as in the first session. Not much happened because F1, Ctrl-F12 and € are not mentioned in yaze.ktt so nothing was reported. There was one interesting event, though. After two consecutive ESC keystrokes, the keytest program switched the translate module into a verbose mode wherein it displays the keycode assembled out of the multi-byte sequence before it is processed. The next session is more interesting ...

```
./keytest sample
```

```
Press keys.  <esc> <esc> <esc> to end
F12Ctrl F12<0D>
Euro symbol<0D>
<esc><esc>
key = 0000007F
<08>
key = 800020AC
Euro symbol
key = 0000001B
<esc>
key = 0000001B
<esc><esc>
```

Sample.ktt has (test) entries for F1, Control-F12 and € so this time something was displayed when those keys were pressed. ESC ESC put the program into verbose mode and then pressing the Backspace key showed that it was generating DEL which was then being translated to BS. Pressing the € key again showed that the UTF-8 sequence <E2><82><AC> (from the first session) was being assembled to Unicode point 20AC which of course came out of the translate routine as "Euro symbol".

Let us back up a little. Run keytest in reporting mode, press a whole bunch of keys and see what happens. You need to interpret the results so you can generate key descriptors.

Usually the cursor keys generate sequences starting with an ESC character followed by an open bracket. Your findings should be fairly similar to mine:

Key	Response
Up	<esc>[A
Down	<esc>[B
Right	<esc>[C
Left	<esc>[D

These, together with Ins, Del, Home, End, PgUp and PgDn are easy to handle because the translation system knows the generated sequences and you can deal with the keys by name. You can't specify that you want to translate the sequence <esc>[A to ^R (for example) but you can just tell the translator to deliver ^R when you press the Up key. Only if you find that the keys are not recognised would you ever have to get into the business of making up new names and key identifiers.

Using the control key in conjunction with the cursor keys yields more useful results such as:

Ctrl Up	<esc>[1;5A
Ctrl Down	<esc>[1;5B

but again you can use the key names because these are known sequences. And you can go on with other modifier keys like Alt, Shift, Alt Shift, Ctrl Shift but the story is the same; you don't need to deal with the actual sequences, just use the qualified key names.

The recogniser does know about single characters and on the main keyboard in combination with the shift keys. Of the shift keys, Alt will present several more translatable names. Some may be intercepted by the operating system before they get to the keytest program but others will sneak through.

Alt q	<esc>q
Alt Q	<esc>Q
Ctrl Alt Q	<esc><11>

Given that ^Q is 11 hex it should be apparent that the general effect of chording with the Alt key is to put an ESC character ahead of the main one. This is a useful finding because now you have a slew of other characters which you could translate if you wanted. Then there are all the top row keys and the function keys to play with. You should find plenty of keys to remap for CP/M without needing to get into the business of altering the keystroke recogniser state machine. The difficulty will be making choices that work well for you and your (CP/M) software.

This is already sufficient. You could stop reading now, go and write one or more key translation tables and start using them. For convenience you should call the main table yaze.ktt and pick any names you like for the others. Remember to keep the file names in lower case. The files need to be in the same directory from which you run yaze-ag. There are no fancy environment settings or search paths.

I get away with a single translation table file (yaze.ktt) because all the software that I use regularly is happy with what I have put into my version of that file. If you have special needs then it is easy to switch tables from inside CP/M:

```
C5:>ktt canetoad        [unloads yaze.ktt and loads canetoad.ktt]
C5:>warts may june      [run my special program using different key map]
C5:>ktt yaze            [put back the default key translation]
```

Fixing the BDOS

By default, yaze-ag runs CP/M 3. Of course you can run CP/M 2.2 if you want, but why would you? Instead of a standard CP/M+ BDOS, yaze-ag uses Simeon Cran's ZPM3. This is a good thing because ZPM3 provides an extremely useful input history like that in many modern operating systems without the need to run an RSX or similar. Furthermore, because it is inside the operating system it lives in the system bank of memory, doesn't consume valuable TPA and provides history service to applications as well as the CCP. However the implementation was a bit quirky. To recall the previous command you had to ask for the previous one and vice versa and one of the keystrokes used (^W) did not match the corresponding function in WordStar and similar text editors (^X).

To fix these issues it was necessary to alter the keystroke dispatch table in ZPM3 itself.

ZPM3 edit function	Original key	New key
previous command	^W	^E
next command	^E	^X
clear to start of line	^X	^W

ZPM3 is just a BDOS workalike. There is nothing sacrosanct or historically precious about it and there was no reason to retain the awkward key assignments. However if you really want Simeon Cran's originals then you can use the old ZPM3 which is provided with this distribution. Simply issue the following commands before starting yaze-ag ...

```
mv yaze-cpm3.boot yaze-cpm3.new
mv yaze-cpm3.cran yaze-cpm3.boot
```

APPENDIX 1 – Key descriptors

These are the names of the keys recognised as candidates for translation ...

Function keys:

- F1, F2, ... F12

Cursor controls:

- up, down, left, right, home, end
- PgUp, PageUp
- PgDn, PageDown

Miscellaneous

- insert, ins
- delete, del
- escape, esc
- quote

Modifiers:

- shift
- control, ctrl
- alt
- literal, lit