berrange.com

Daniel P. Berrangé » Blog Archive

Daniel Berrange

9-12 minutes

In learning about virtual keyboard handling for my previous post, I spent alot of time researching just what scan code & key code sets are used at each point in the stack. I trawled across many websites and source code trees to discover this information, so I figure I should write it down in case I forget it all in 12 months time when I next get a keyboard handling bug report. I can't guarantee I've got all the details here correct, so if you spot mistakes, leave a comment.

IBM XT keyboard

In the beginning the PC was invented. This has made a lot of people very angry and has been widely regarded as a bad move. The IBM XT scan codes are first set encountered. Most of the scan codes were a just a single byte, with the high bit clear for a make event and set for a break event. The keycodes don't correspond at all to ASCII values, rather they were assigned incrementally across rows. As an example, the key in the 4th row, second column (labelled A on a us layout) has value 0x1e for make code and 0x9e for break code. Keys on the numeric keypad would generate multi-byte sequences, where the leading byte is "0xe0" (sometimes referred to as the grey code, I guess because the keypad was a darker grey plastic)

IBM AT & PS/2 keyboards

The IBM AT keyboard was incompatible with the XT at a wire protocol level. In terms of scan codes though, it can support upto 3 sets. The first set is the original XT scan codes. The second set is a new AT scan codes. The third set is so called PS/2 scan codes. In all cases though, the second set is the default and the first & third sets may not even be supported. There is no resemblance between AT and XT scan codes. The use of the high bit for break code was abandoned. The same scan code is used by make and break, but in the break case it is prefixed by a byte 0xf0. As an example, the key in the 4th row, second column (labelled A on a us layout) has the value 0x1c. Again the keys on the numeric keyboard generated multi-byte sequences with a leading “0xe0”.

For added fun, regardless of what scan code set the keyboard is generating the i8042 keyboard controller will, by default, provide the OS with XT scan codes unless told otherwise. This was for backwards compatible with software which only knows about the previous XT scan code set.

Over time Microsoft introduced the Windows keys, then Internet keys arrived followed by all sorts of other function keys on laptops. The extended 0xe0 (“grey code”) sequence space was used for these new scan codes. Of course these keys needed scan codes in both the XT and AT sets to be defined. So for example, the left Windows key got the XT make/break scan codes 0xe0+0x5c and 0xe0+0xdc and the AT make/break scan codes 0xe0+0x27 and 0xe0+0xf0+0x27.

USB keyboards

The introduction of USB input devices, allowed for a clean break with the past, so XT/AT scan codes are no more in the USB world. The USB HID spec defines a standard 16 bit scan code set for all compliant keyboards to use. The USB HID spec gives scan codes names based on the US ASCII key cap labels. Thus the US ASCII key labelled ‘A’ has HID page ID 0x07 and HID usage ID 0x04. Where both the XT/AT sets and USB HID sets support a common key it is possible to perform a lossless mapping in either direction. There are, however, some keys in the USB HID scan code set for which there isn’t a XT/AT scan code.

Linux internals

The Linux internal event subsystem has defined a standard set of key codes that are hardware independant, able to represent any scan code from any type of keyboard whether AT, XT or USB. There are names assigned to the key codes based on the common US ASCII key cap labels. The key codes are defined in /usr/include/linux /input.h. For an example ‘#define KEY_A 30’. For convenience, key codes 0-88 map directly to XT scan codes 0-88. All the low level hardware drivers for AT, XT, USB keyboards, etc have to translate from scan codes to the Linux key codes when queueing key events for dispatch by the input subsystem. In “struct input_event”, the “code” field contains the key code while the “value” field indicates the state (press/release). Mouse buttons are also represented as key codes in this set.

Linux console

The Linux console (eg /dev/console or /dev/tty*) can operate in several different modes.

- RAW: turns linux keycodes back into XT scancodes
- MEDIUMRAW: an encoding of the linux key codes. key codes < 127 generated directly, with high bit set for a release event. key codes >= 127 are generated as multi-byte sequences. The leading byte is 0x0 or 0x1 depending on whether a key press or release. The second byte contains the low 7 bits and third byte contains the high 7 bits. Both second and third bytes always have the 8th bit set. This allows room for future expansion upto 16384 different key codes
- ASCII: the ascii value associated with the key code + current modifiers, according to the loaded keymap
- UNICODE: the UTF-8 byte sequence associated with the key code + current modifiers, according to the loaded keymap

The state of the current console can be seen and changed using the “kbd_mode” command line tool. Changing the mode is not recommended except to switch between ASCII & UNICODE modes

Linux evdev

The evdev driver is a new way of exposing input events to userspace, bypassing the traditional console/tty devices. The “struct input_event” data is provided directly to userspace without any magic encodings. Thus apps using evdev will always be operating with the Linux key code set initially. They can of course convert this into the XT scan codes if desired, in exactly the same way that the console device already does when in RAW mode.

Xorg XKB

In modern Xorg servers, XKB is in charge of all keyboard event handling. Unusually, XKB does not actually define any standard key code set. At least not numerically. In the XKB configuration files there are logical names for every key, based on their position. Some names may be obvious ‘<TAB>’, while others are purely reflecting a physical row/column location ‘<AE01>’. Each Xorg keyboard driver is free to define whatever key code set it desires. The driver must provide a mapping from its key code values to the XKB key code names (in /usr/share/X11/xkb/keycodes/). There is then a mapping from XKB key code names to key symbols (in /usr/share/X11/xkb /keysyms). The problem with this is that the key code seen in the XKeyEvent can come from an arbitrary set of which the application is not explicitly aware. Three common key code sets from Xorg will now be mentioned.

Xorg kbd driver

The traditional "kbd" driver uses the traditional Linux console/tty devices for receiving input, configuring the devices in RAW mode. Thus it is accepting XT scan codes initially. The kdb keycodes are defined in its source code at src/atKeynames.h For scan codes below 89, a key code is formed simply by adding 8. Scan codes above that have a set of re-mapping rules that can't be quickly illustrated here. The mapping is reverseable though, given a suitable mapping table.

Xorg evdev driver

The new "evdev" driver of course uses the new Linux evdev devices for receiving input, as Linux key codes. It forms its scan codes simply by adding 8 to the Linux key code. This is trivially reverseable.

Xorg XQuartz driver

The Xorg build for OS-X, known as XQuartz, does not use either of the previously mentioned drivers. Instead it has a OS-X specific keyboard driver that receives input directly from the OS-X graphical toolkit. It receives OS-X virtual key codes and produces X key codes simply by adding 8. This is again trivially reversable.

RFB protocol extended key event

The RFB protocol extension defined by GTK-VNC and QEMU allows for providing a key code in addition to the key symbol for all keypress/release events sent over the wire. The key codes are defined to be based on a simple encoding of XT scan code set. Single byte XT scan codes 0-127 are sent directly. Two-byte scan codes with the grey code (0xe0) are sent as single bytes with the high bit set. eg 0x1e is sent as 0x1e, while 0xe0+0x1e is sent as 0x9e (ie 0x1e | 0x80). The press/release state is sent as a separate value.

QEMU internals

QEMU's internal input subsystem for keyboard events uses raw XT scan codes directly. The hardware device emulation maps to other scan code sets as required. It is no coincidence that the RFB protocol extension for sending key codes is a trivial mapping to QEMU scan codes.

Microsoft Windows

The Windows GDI defines a standard set of virtual key codes that are independent of any particular keyboard hardware.

Apple OS-X

OS-X defines a standard set of virtual key codes that are based on the original Apple extended keyboard scan codes.

SDL

The SDL_KeyboardEvent struct contains a “scancode” field. This provides the operating system dependant key code corresponding to the key event. This is easily interpreted on OS-X/Windows, but on Linux X11 this requires knowledge of what keyboard driver is used by Xorg.

GTK/GDK

The GDKKeyEvent struct contains a “hardware_keycode” field. This provides the operating system dependant key code corresponding to the key event. This is easily interpreted on OS-X/Windows, but on Linux X11 this requires knowledge of what keyboard driver is used by Xorg.

Future ideas

The number of different scan code and key code sets is really impressive / depressing / horrific. The good news is that given the right information, it is possible to map between them all fairly easily, in a lossless fashion. The bad news is that the data for performing such mappings is hidden across many web pages and many, many source trees. It would be great to have a single point of reference providing all the scan/key code sets, along with a tool that can generate the mapping tables & example code to convert between any 2 sets.