Kirk 2.01

Kernel driver for Infra-Red Keyboard
A driver for a keyboard talking on the serial port
October 2001

by Alessandro Rubini (`rubini@linux.it`)

This file documents version 2.01 of the Kirk device driver and associated user-space programs (October 2001).

The driver implements a line discipline for tty devices and is actually not involved with any infra-red. The target device of this driver is a Sejin peripheral that connects to a standard serial port. The device receives input from a keyboard (possibly with a built-in pointer) and a remote control device. Version 2.00 adds support for version 2.4 of the kernel and for the internal mouse (that is builtin in the keyboard). The mouse is fed to user space via a misc device that speaks the PS2 protocol.

1 Compiling and Loading

To compile the driver and associated code run `make`. This compiles '`kirk.o`' kernel module and the user-space utilities called '`kirkdump`' and '`kirkrun`'. By default the '`Makefile`' assumes kernel headers live under '`/usr/src/linux/include`'; if yours are found in a different place set `KERNELDIR` in your environment or on the command line of `make`. For example:

```
make KERNELDIR=/usr/src/linux-2.2
```

To load the device driver you just need to run '`insmod`' or '`modprobe`', with '`kirk.o`' as argument. The driver registers a line discipline for tty devices. Activation of the line discipline must be performed by a user-space program.

Please note that support for `<linux/modversions.h>` has beed disabled after version 2.00 of *kirk* because I had a mismatch when loading on the lixux-2.2.12-20 as precompiled by RedHat, so I'd better compile without version support and avoid load-time errors.

2 Features of the driver

The driver has been designed to be as flexible as possible, especially with regard to possible changes of the key layout. This means that the protocol used by the actual device to report key-press and key-release events is somewhat configurable at compile time (instead of being completely hardwired in source code).

kirk is also able to drive keyboard leds (by writing commands to the serial port), but this feature only works with 2.2.18 and later kernels, as the internals were not exported with earlier kernel versions (2.2.18 added it because it was needed in supporting USB keyboards).

2.1 Keymap Definition

The source file '`keymap.c.in`' is used to generate the output file '`keymap.c`', which is then fed to the compiler. The initial part of '`keymap.c`' is preserved in the process, so you should not delete '`keymap.c`', which acts as both input and output in the build process.

The length of data packets is fixed to four bytes, and '`keymap.c.in`' defines the meaning of the packet. The program, however, has hardwired assumptions on the overall format of the packet: the first byte is assumed to be a "magic" identifier, the second byte is an "input

device" selector (so several device can talk through the same serial port) and the fourth byte is the key identifier. The most significant bit of the third byte is ignored and other bits are ignored.

Each line of the '`keymap.c.in`' input file is processed separately; blank lines and lines that begin with an hash mark (`#`) are ignored, other lines are interpreted according to the first blank-separated word:

`MAGIC`

This line defines the value that is expected in the first data byte, the value is parsed as an integer. All valid packets must begin with this byte value. Only one such lines must appear in the file.

`MAP`

This line is used to select a specific input source for the following `I` and `O` lines. The `MAP` keyword is followed by a single byte value.

`I`

This line lists a number of "input" keycodes. Data appearing after the leading `I` is parsed as a list of byte values, each value representing one of the possible values in the fourth byte of a data packet.

`O`

The line lists "output" keycodes. Data appearing after the leading `O` is parsed as a list of byte values, and it must be the same length as the preceding `I` line. Each value represents the PC-keyboard keycodes that must be associated to the corresponding "input" keycode. By a proper layout of the `I` and `O` lines, you can make the file visually similar to the keyboard map, thus greatly simplifying the identification of any error in defining the keys.

`LED`

Lines with a leading `LED` string are used to specify how output leds are controlled. Each line is made up of three words: `LED`, a sub-identifier and a byte value. The '`base`' sub-id is used to state the output byte that means "no led", the '`scroll`', '`num`', and '`caps`' sub-ids specified a value (usually a single bit) that is XOR'd to the base value in order to light the specified led.

`MOUSE`

Lines with a leading `MOUSE` string are used to specify how mouse data packets are decoded. The directives specify a set of keymaps (as mouse buttons are reported as bits in the first byte, the one that identifies the keymap) that must be managed by the mouse decoder. The values are an AND-mask that is applied to the first data byte after the magic number and the value that is expected as result.

Other `MOUSE` lines specify how buttons and coordinates are decoded. Either kind of line includes an identifier (1-3 for buttons and x or y for coordinates) and three numbers. The numbers represent the data byte to be used, an AND-mask and an XOR-mask. Using the XOR value it is possible to invert the coordinates or button state.

The best exemplification of the file format is a working example; '`keymap.c.in`', as distributed, is a good example and is quite well commented as well. The following lines are extracted from the distributed '`keymap.c.in`', stripping most data and all the comments:

```
# comment
MAGIC 0x97
MAP 0x10

I       0x08              0x1B
O        89                90

I       0x59     0x69     0x79
O        79       80       81

MAP 0

I  0x0c 0x41
O  111  111

LED base 0xe0

LED scroll 0x01
LED num    0x02
LED caps   0x04

MOUSE maps 0xf8 0xf8

MOUSE button 1    1   0x01  0x00
MOUSE button 2    1   0x02  0x00
MOUSE button 3    1   0x04  0x00

MOUSE motion x    2   0xff  0x00
MOUSE motion y    3   0xff  0x00
```

2.2 Led support

The serial device is assumed to turn on and off keyboard leds by means of single-byte commands (where individual bits select individual leds).

The actual values to be output are defined in '`keymap.c.in`', whose syntax is described in Section 2.1 [Keymap Definition], page 1.

Since the Linux kernel sources earlier than 2.2.18 don't export support for keyboard leds to modules, *kirk* won't have such support when compiled against earlier kernel versions.

2.3 Autorepeat

Key-repeat data packets sent by the hardware are disregarded, as they make no practical sense (moreover, the way they are generated requires extra software overhead in order to remove false key-press events. The driver keeps track of which keys are currently down in order to properly remove all repeat events sent by hardware (instead of just dropping a packet that is identical to the previous packet, which would work if auto-repeat was implemented sanely in hardware).

The driver generates its own auto-repeat events by using kernel timers. Auto-repeat is characterized by two time lapses: the delay before the first repeat event is generated and the delay across successive repeat events. Both time lapses can be specified as a count of clock ticks (which, on the x86 platform, is once every 10ms).

The default auto-repeat values are `50` (half a second of delay) and `10` (100 ms across each repeat event). They can be changed by writing to '`/proc/sys/dev/kirk`'. For example, this command can be used to have a delay of 1 second and one repeat every 200 ms:

```
# echo "100 20" > /proc/sys/dev/kirk
```

and the following command lowers the delay to 200ms and a repeat rate of 33 chars per second:

```
# echo "20 3" > /proc/sys/dev/kirk
```

Only the superuser is allowed to change the time lapses.

2.4 Mouse Support

Version 2.00 of the package adds mouse support as a misc device. The device entry point uses a dynamic minor number and appears in '`/proc/misc`' as *kirk*. Decoding of mouse data packets is configured in '`keymap.c.in`' (see Section 2.1 [Keymap Definition], page 1), and the information is reported to user space as PS/2 data packets. Both X and gpm can run flawlessly using that device.

The '`gpm`' command like should be something like:

```
gpm -m /dev/kirkmouse -t ps2
```

while the configuration for *XFree-3.3* should look like:

```
Section "Pointer"
Protocol        "PS/2"
Device          "/dev/kirk"
Buttons         2
Emulate3Buttons
Emulate3Timeout 50
EndSection
```

For *XFree-4.0* the configuration will look like:

```
Section "InputDevice"
Identifier  "Mouse1"
Driver      "mouse"
Option "Protocol"    "PS/2"
Option "Device"      "/dev/kirk"
Option "Emulate3Buttons"
```

```
Option "Emulate3Timeout"    "50"
EndSection
```

3 User-Space Programs

The package includes two user-space programs. One for dumping data bytes from the serial port and one for actually running the line discipline and generate keyboard events.

3.1 kirk_load and kirk_unload

These are two trivial scripts that can be used to load and unload the module.

In addition to '`insmod`' and '`rmmod`', respectively, the files also create and remove the mouse device entry point as '`/dev/kirkmouse`'.

3.2 kirkdump

The program is used to print to `stdout` data bytes received by the serial port, four at a time. It doesn't use the *kirk* line discipline but normal serial communication (with the default line discipline).

It receives a single command line argument, the name of the serial device to open. For example:

```
kirkdump /dev/ttyS0
```

3.3 kirkrun

The program activates the *kirk* line discipline on a serial port. It needs the '`kirk`' module to be loaded. To restore the default line discipline (and thus stop *kirk* operation), you can simply kill the program (as the default line discipline is restored when a tty device is closed).

The program receives a single command line argument, the name of the serial device to open. For example:

```
kirkdump /dev/ttyS0
```

Table of Contents