

Perl script controls toy USB canon

# REPLACING TIN SOLDIERS

Although a USB toy such as a polystyrene rocket launcher only includes a Windows CD, it works fine on Linux with a spot of reverse engineering. With libusb, this doesn't even require compiling a device driver – Perl controls the device from userspace. **BY MICHAEL SCHILLI**

**T**rouble at the office? It doesn't necessarily need to lead to a battle, such as in the video *The Great Office War* [2] by toy manufacturer Hasbro. Even if you don't have an attack plan, the USB-controlled Rocket Baby rocket launcher (Figure 1), by Chinese manufacturer Cheeky Dream, is a bargain at less than US\$ 20. Not only did it cheer up my colleagues at work, it also



Figure 1: The USB rocket launcher Rocket Baby, by Cheeky Dream.

gave me an opportunity to study the Linux kernel's fairly complex USB subsystem [3].

Opening the box reveals a CD for Windows XP, but no trace of a Linux driver. This seems to have provoked a number of gadget fans in the developer community to investigate the USB protocol the toy uses on Windows with USB sniffers such as USBsniff, to reverse engineer the interfaces, and create bindings for languages such as Python, or even for completely different operating systems [4].

When the toy is plugged into an empty USB slot, the Ubuntu Hardy Heron distri-

bution autodetects it. The kernel messages, which can be read in the `/var/log/messages` logfile (Figure 2), tell you that the toy rocket launcher is now connected to the Intel-based PC's UHCI controller.

According to the logfile, the kernel's USB subsystem has mapped the rocket launcher to `usb 5-1`. The sysfs tree below `/sys/bus/usb/devices/5-1` gives you the details. The USB filesystem, `usbfs`, projects the kernel's internal USB data to userspace this way. Figure 3 shows that the `idVendor` value for the launcher is `0x0a81`, and the `idProduct` is `0x0701`, as can be determined easily by looking up

```
mschilli@mybox:/mnt/data/big1/mschilli.do.not.delete/DEV/articles/rocket/eg
Mar  1 10:23:06 mybox kernel: [404953.755026] usb 5-1: new low speed USB device
using uhci_hcd and address 4
Mar  1 10:23:06 mybox kernel: [404953.987072] usb 5-1: configuration #1 chosen f
rom 1 choice
Mar  1 10:23:06 mybox kernel: [404954.017977] hiddev97hidraw5: USB HID v1.00 Dev
ice [Rocket Baby Rocket Baby] on usb-0000:00:1d.1-1
```

Figure 2: After you plug in the rocket launcher, the kernel detects the device and assigns a USB entry to it.

the content of the respective files in the `sysfs` tree. According to the order in which you plug in the USB devices, the kernel assigns varying USB numbers for them; instead of `usb 5-1`, it could be `usb 3-1` next time.

This said, there is only one device with the `idVendor` and `idProduct` values we just discovered plugged into the PC; thus, a program can find the USB address of the device reliably and fairly quickly by parsing the USB tree until it finds the right combination.

Linux normally uses kernel device drivers to talk to USB devices. A driver is difficult to program because there is no safety net, as taken for granted in user space; the slightest pointer error will torpedo the whole Linux system and force a reboot. On top of this, users would need to recompile the device driver for each new kernel and load the module as root by running `modprobe`. Data structures tend to change rapidly in the kernel, and it is possible that the source code you write for kernel 2.6.22 goes out the window with version 2.6.24.

But if you do not need high data throughput or realtime responses, there is no need to leave the control logic with the kernel. Instead, the kernel has the `usbfs` that lets you talk to USB devices at the hardware level, and this means that you can implement the driver in user-space.

The Open Source `libusb` project [5] provides a convenient library for C programs, and the Perl `Device::USB` module

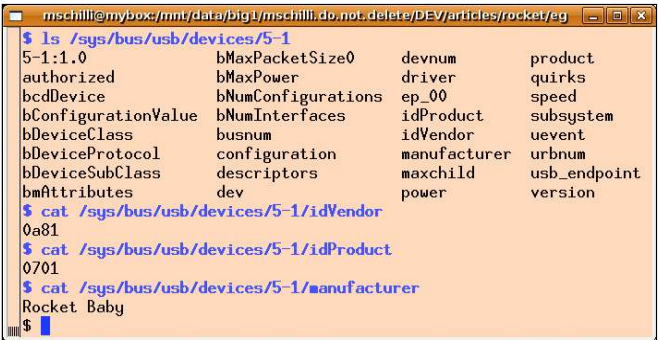
from CPAN wraps Perl functions around it.

Listing 3 shows you how to raise the rocket launcher's barrels by about half an inch with just a couple of lines of Perl code. First, the `find_device()` function uses the

`Device::USB` module to locate the device with the `idVendor` and `idProduct` values discovered beforehand in the USB tree. If this succeeds, the `open()` method opens a connection to the device.

The kernel's USB subsystem supports four different communication modes for USB controllers: *Control Transfers* for short messages, *Bulk Transfers* for larger volumes of data, *Interrupt Transfers* for time-critical data, and *Isosynchronous Transfers* for realtime data. Reverse engineering helped the developers discover that the rocket launcher uses one-byte control messages to move the tower and fire the polystyrene rockets. Listing 1 shows the codes for various actions.

One code moves the launcher until another code either changes the direction or a stop command cancels the movement, which is important because if a program starts a movement and then fails, the rocket launcher's motor will annoyingly keep on running. The hex values passed into the `control_msg()`



```

mschilli@mybox:/mnt/data/big1/mschilli.do.not.delete/DEV/articles/rocket/eg
$ ls /sys/bus/usb/devices/5-1
5-1:1.0          bMaxPacketSize0  devnum          product
authorized      bMaxPower        driver          quirks
bcdDevice        bNumConfigurations ep_00           speed
bConfigurationValue bNumInterfaces  idProduct      subsystem
bDeviceClass     busnum           idVendor       uevent
bDeviceProtocol  configuration    manufacturer   urbnum
bDeviceSubClass  descriptors      maxchild       usb_endpoint
bmAttributes     dev             power          version
$ cat /sys/bus/usb/devices/5-1/idVendor
0a81
$ cat /sys/bus/usb/devices/5-1/idProduct
0701
$ cat /sys/bus/usb/devices/5-1/manufacturer
Rocket Baby
$

```

Figure 3: Linux shows details of hotplugged devices in the `/sys` tree.

method in Lines 14 and 22 define how the USB interface passes the control byte on to the controller: `0x21` stands for the request type, `0x09` for `USB_REQ_SET_CONFIGURATION`, `0x02` for `USB_RECIP_ENDPOINT`, and the value `0` for an unused index. Then comes the control code (`0x02` for moving the barrels up in line 15) for driving the launcher. Perl's `chr()` function transforms an integer value, such as `0x02`, into a single byte containing the same value.

The last two parameters specify the length of the string, `1`, or exactly one byte in our case, and the response wait time in milliseconds (`1000`) before the program times out.

After this, the test program takes a short break of a tenth of a second (`10,000` microseconds) thanks to the CPAN `Time::HiRes` module and its `usleep()` function, before going on to send the `0x20` control byte, which the receiving end interprets as a stop command, thus switching the rocket

### Listing 1: Control Codes

```

01 down    0x01,
02 up      0x02,
03 left    0x04,
04 right   0x08,
05 fire    0x10,
06 stop    0x20,
07 start   0x40,

```

### Listing 2: Parameters for control\_msg()

```

01 $requesttype => 0x21
02 $request     => 0x09
03 $value       => 0x02
04 $index       => 0
05 $bytes       => chr(...)
06 $size        => 1
07 $timeout     => 1000

```

### Listing 3: rocket-test

```

01 #!/usr/local/bin/perl -w          19
02 use strict;                        20 # Stop
03                                    21 $val = 0x20;
04 use Time::HiRes qw(usleep);        22 $dev->control_msg( 0x21,
05 use Device::USB;                   23 0x09, 0x02, 0, chr($val),
06 my $usb = Device::USB->new;         24 1, 1000 );
07 my $dev =                            25
08 $usb->find_device( 0xA81,           26 # Read status
09 0x701 );                             27 $val = 0x40;
10 $dev->open;                           28 my $buf;
11                                       29 $dev->control_msg( 0x21,
12 # Move Up                             30 0x09, 0x02, 0, chr($val),
13 my $val = 0x02;                       31 1, 1000 );
14 $dev->control_msg( 0x21,              32 $dev->bulk_read( 1,
15 0x09, 0x02, 0, chr($val),           33 $buf = "", 1, 1000 );
16 1, 1000 );                            34 printf "Status %08b\n",
17                                       35 ord($buf);
18 usleep(150_000);

```



launcher tower motor off. In Listing 3, the two calls to `control_msg()` thus move the launcher tower up for a tenth of a second. If the tower is not at its maximum elevation already, this means that you hear the motor for a fraction of the second and the polystyrene rockets are elevated by about 20 degrees.

## Fire!

When you fire the rockets, note that the motor needs to pump for about two seconds to build up the pressure needed to fire the projectiles. So that the program can discover when to switch off the motor, because the rocket has been released, it must access the USB interface and read the rocket launcher controller data. The controller reports which actions are available now, and which are not. If the tower is swiveled as far right as it will turn, the controller returns a status string with a value of `0x08` (binary `0000_1000`) to show that all actions apart

from `0x08` are now available; as you will see in box 1, `0x08` represents the direction *right*. If the tower is swiveled as far left and to the bottom as it will turn, the controller returns a status message of `0x05` (binary `0000_0101`), because both `0x01` (*down*) and `0x04` (*left*) are now blocked. In a similar fashion, the USB device sets a `0x10` flag (binary `0001_0000`) shortly after firing to tell the controller that it can now issue a `0x20` to switch off the motor, unless you want to fire the next in line of the total of three rockets.

To check the launcher status, the controller first sends a control code of `0x40` via `control_msg()` to the USB device, which is immediately followed by a bulk transfer using the `bulk_read()` method to pick up the data string returned by the device. Line 34 in Listing 3 writes the result, which is `00000000` in most cases, unless the tower is swiveled to one of the limits, or at minimum or maximum elevation, or a rocket has just been fired.

### Listing 4: center-fire

```

01 #!/usr/local/bin/perl -w          32
02 use strict;                      33 do_until( "left",
03                                  34 $right_elapsed / 2 );
04 use                                35 do_until( "down",
05 Device::USB::MissileLauncher::RocketBaby; 36 $up_elapsed / 2 );
06 use Time::HiRes                  37
07 qw(usleep gettimeofday          38 for ( 1 .. 3 ) {
08 tv_interval);                   39 do_until("fire");
09                                  40 usleep(100_000);
10 my $rb =                          41 }
11 Device::USB::MissileLauncher::RocketBaby 42
12 ->new();                          43 #####
13                                  44 sub do_until {
14 do_until("left");                45 #####
15 do_until("down");                46 my ( $what, $max_time ) =
16                                  47 @_;
17 my $right_start =                 48
18 [gettimeofday];                   49 my $start = [gettimeofday];
19 do_until("right");                50
20 my $right_elapsed =                 51 while ( $rb->cando($what) )
21 tv_interval(                       52 {
22 $right_start,                       53 $rb->do($what);
23 [gettimeofday]                     54 usleep(100_000);
24 );                                   55 last
25                                     56 if defined $max_time
26 my $up_start =                       57 and
27 [gettimeofday];                     58 tv_interval( $start,
28 do_until("up");                       59 [gettimeofday] ) >
29 my $up_elapsed =                       60 $max_time;
30 tv_interval( $up_start,               61 }
31 [gettimeofday] );                   62 $rb->do("stop");
                                         63 }

```

The `Device::USB::MissileLauncher::RocketBaby` module from CPAN provides a neat interface abstraction; a newly constructed object offers the `do()` and `cando()` methods, which expect actions as strings such as *left*, *up*, *fire*, or *stop*. The `do()` method executes these actions, whereas `cando()` just issues a status request and a bulk read to check whether an action is currently available.

Listing 4 shows you how to use this. To start, it rotates the tower down and to the left to the limits to ascertain the precise position, then it measures the time required to elevate the tower completely and to swivel it fully to the right. Then it divides the two times by half and centers the tower using the calculated values before firing all three rockets one after another.

## Installation

Linux needs the `libusb-dev` package in userspace to access USB devices. Any fairly recent distribution will have this. The `Device::USB` and `Device::USB::MissileLauncher::RocketBaby` modules are best installed using a CPAN shell. Various types of rocket launchers use different code combinations; when in doubt, you can search for the right combination on the Internet and then wrap it up in an abstraction such as the CPAN `RocketBaby` module.

A Youtube video [6] shows you how the rocket launcher responds to the *center-fire* script. And now a message from the Homeland Security Department: please avoid exporting this script to untrusted countries. ■

## INFO

- [1] Listings for this article: [http://www.linux-magazine.com/resources/article\\_code](http://www.linux-magazine.com/resources/article_code)
- [2] The Great Office War: <http://www.youtube.com/watch?v=pVKnF26qFFM>
- [3] *Essential Linux Device Drivers*, Sreekrishnan Venkateswaran, Prentice Hall, 2007
- [4] *Python Interfacing a USB Missile Launcher*, Pedram Amini: <http://dvlabs.tippingpoint.com/blog/2009/02/12/python-interfacing-a-usb-missile-launcher>
- [5] The libusb project: <http://libusb.sourceforge.net>
- [6] Youtube video: <http://www.youtube.com/watch?v=6qTRhDijJc>