NUI Chapter 4. A Motion-tracking Missile Launcher

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/jg/??; only important fragments are described here.]

Protect your cubicle with a lethal defensive shield, deterring foolish intruders by firing missiles at them as they approach. In order to save money, I won't be using state-of-the-art tech., such as the SS-27 Sickle B ballistic system, but the somewhat cheaper Dream Cheeky launcher (http://www.dreamcheeky.com/). This extremely dangerous weapon is sometimes disparaged as a mere toy, perhaps because it is. But those naysayers don't realize that I'm combining the launcher's awesome capabilities with the motion detection code from chapter 3.

When movement is detected, the launcher automatically rotates left, right, up or down to point at the target. If the motion continues, then the launcher fires one of its missiles. This stupendous weapon system is illustrated in Figure 1, attacking its lowly creator.

Figure 1. The Motion-tracking Missile Launcher.

The main topic of this chapter is how to write the launcher's control software, which operates via USB. The tricky part is that the Dream Cheeky gadget doesn't come with a developers API or any technical documentation, and so I'll use USB monitoring software, and some detective work, to develop one myself. The other problem is that Java doesn't support USB I/O as standard, so I need to utilize third-party libraries.

The good news is that much of this work can be applied to developing interfaces for other 'fun' USB devices, which typically use the same kinds of control and interrupt messaging.

1. Playing with the Missile Launcher

The missile launcher comes from Dream Cheeky (http://www.dreamcheeky.com/), which specializes in unusual USB gadgets for the PC, such as a roll-up piano keyboard, drum kit, and webmail notifier postbox. The launcher shown in Figure 1 (called the MSN Missile Launcher) is a few years old, and the device was updated in 2010, and renamed as the Storm O.I.C. (Over Internet Control) (http://www.dreamcheeky.com/storm-oic-missile-launcher). The launcher plugs into a PC via a standard USB cable, and also comes with a webcam with its own USB connector (that's why there are two leads going into the PC shown in Figure 1). There are also launcher versions without a webcam, which are much less fun.

Sadly, the company no longer sells the MSN, although its GUI control software can still be downloaded from the Dream Cheeky website, and the gadget is available from other suppliers..

Dream Cheeky doesn't aim its products at developers (there's no APIs or technical info included with the devices), focusing instead on making them fun and easy to use via slightly cheesy GUIs. Figure 2 shows the interface for the MSN launcher.

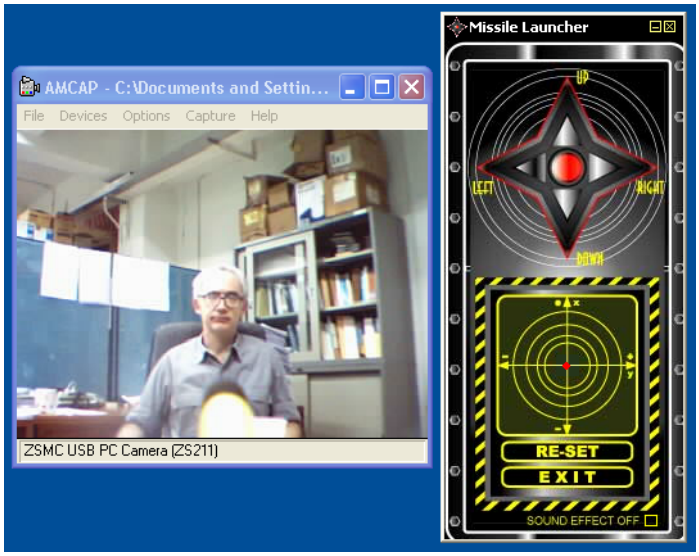

Figure 2. The MSN Missile Launcher GUI.

Two applications need to be started – a webcam display window and a Missile Launcher control panel. The control panel turns the launcher left, right, up, or down, when the user presses the arrows in the star picture. Pressing the red circle in the middle of the star fires a missile. The webcam is attached to the top of the launcher (see Figure 3), so rotates along with it.

Figure 3. The Launcher Viewed from the Front.

The MSN is really two USB devices, and its possible to use the webcam separately from the launcher.

The arrow keys in the GUI of Figure 2 stop working when the launcher is rotated too far in the left, right, up, or down directions. However, there's no limit on the number of times the fire button can be pressed, even though there's only three missiles in the launcher. The missiles are fired in a fixed order in a clockwise order starting from the bottom right.

By the way, when I use "left" and "right", I'm assuming that I'm standing directly behind the launcher (by far the safest spot).

The "RE-SET" button on the GUI moves the launcher back to a default position, facing forwards and pointing horizontally. It does this by turning to its maximum left-most and down-facing positions, and then returning to a default orientation.

2. Where I'm Going with this Chapter

I'll be writing an application called MotionLauncher in this chapter, which is shown in relationship to its third-party libraries in Figure 4.

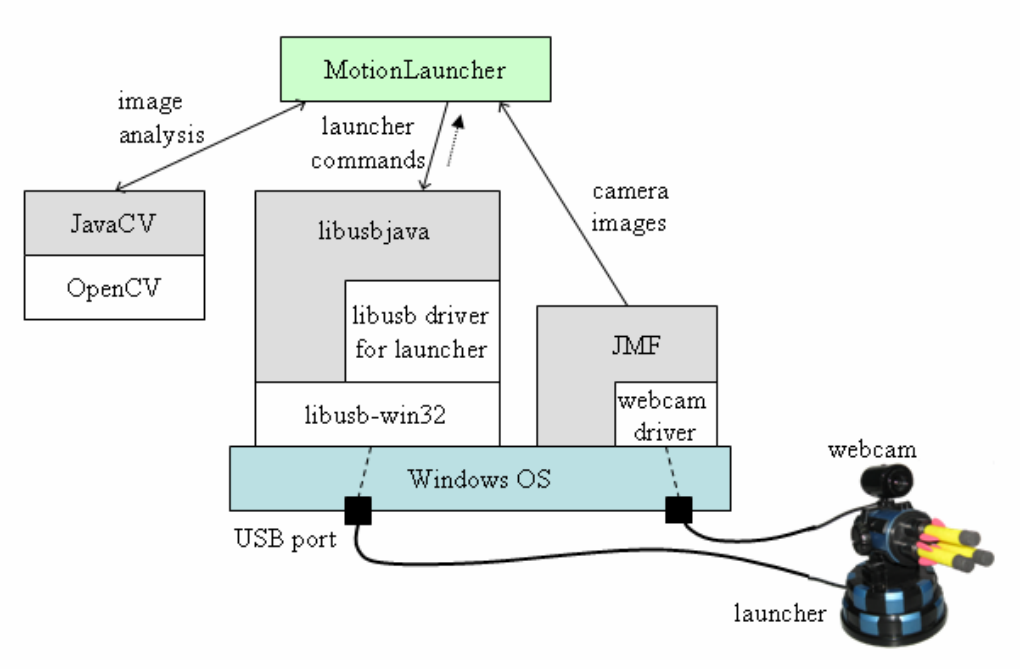

Figure 4. The MotionLauncher Application and its Libraries.

I've already utiilized two of the three library stacks beneath MotionLauncher. The camera imaging and analysis (for detecting movement) are almost unchanged from NUI Chapter 3, so I'll be reusing large chunks of the MotionDetection application. My new code employs the middle stack of Figure 4, which consists of two libraries and a driver.

libusb-win32 (http://sourceforge.net/apps/trac/libusb-win32/wiki) is a port of the libusb 0.1 USB library to MS Windows, which allows applications to access a device without writing a driver. Instead libusb-win32 comes with a tool that generates them, and I'll employ it to create the driver for the launcher shown in Figure 4. libusb-win32 is a C/C++ API, and so I access it via LibusbJava (http://libusbjava.sourceforge.net/).

Before I start writing USB code, I should spend a little time explaining the USB protocol. Also, I'll have to turn investigator to find out the specifics of how the launcher uses USB to communicate with the PC.

3. A Quick Introduction to USB

USB (Universal Serial Bus) communication is carried out between a host and a device. In this chapter, the host is my PC running the MotionLauncher application, and the device is the Dream Cheeky launcher. The host detects devices, manages data flow on the communications bus, carries out error checking, and provides power to the devices.

Due to the wide variety of devices that USB needs to support, the capabilities of a given device are described using a hierarchy of four descriptors, illustrated in Figure 5.

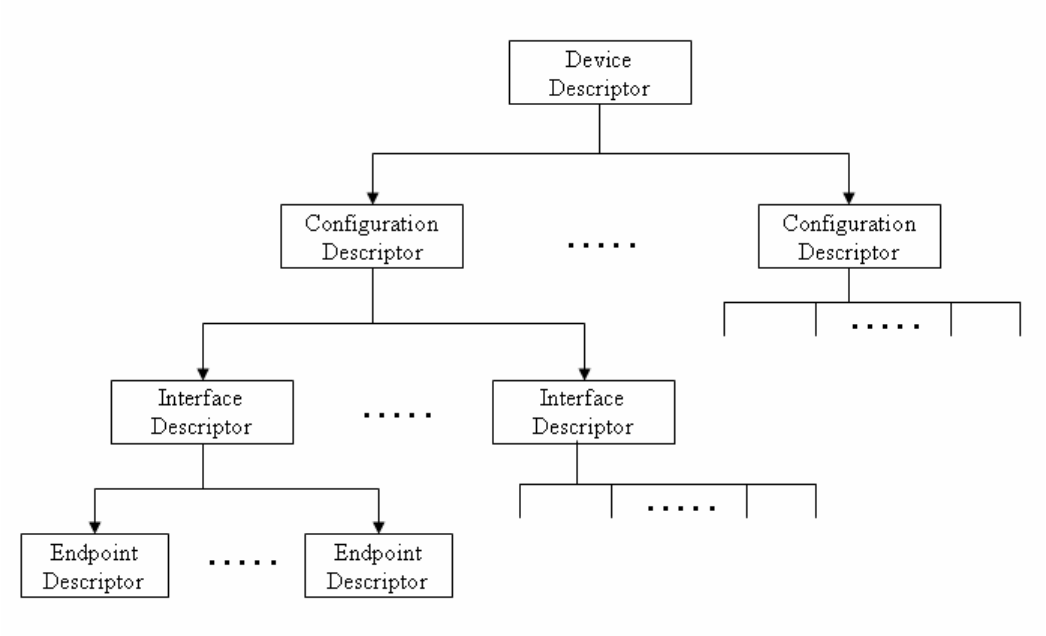

Figure 5. The Descriptor Hierarchy for a USB Device.

The top-most *device* descriptor includes vendor and product IDs (two hexadecimal digits), which uniquely identify the device.

There may be one or more *configuration* descriptors which explain the device's power requirements (e.g. self-powered or drawing energy from the host). Many devices, including all the 'toy' gadgets I've encountered, only have a single configuration descriptor.

A configuration may utilize multiple *interface* descriptors, each one assigned to a particular device feature or function. For example, a combination fax/scanner/printer might have three interface descriptors, one each for the fax, scanner, and printer functionality. An interface descriptor includes device class and protocol information. Most USB toys belong to the Human Interface Device (HID) class, which also includes more common hardware such as keyboards, pointing devices, and game controllers.

A interface descriptor groups together *endpoint* descriptors; each endpoint represents the point where data leaves the host for the device (or enters it). Each endpoint defines the type of data transfer, the direction that data travels, message packet size, and other data details.

USB supports four data transfer types: control, bulk, interrupt, and isochronous. Control transfers are typically used for sending short command messages from the host to the device, and receiving back status details. Bulk transfers are intended for applications where the rate of transfer isn't critical, such as sending a file to a printer. Interrupt transfers are for devices that receive the host's attention periodically, such as keyboards and mice. Isochronous transfers have guaranteed delivery time but no error correcting, so are often utilized for streaming audio and video.

The data transfer direction is defined from the host's perspective: an IN endpoint receives data from the device intended for the host (e.g. the PC) and an OUT endpoint sends data to the device.

Every device has an endpoint 0, configured for control transfers, and there's rarely need for more endpoints in USB toys. This means that a typical gadget only has a single device descriptor, configuration descriptor, interface descriptor, and endpoint descriptor. The Dream Cheeky launcher actually employs two endpoints: endpoint 0 for control transfers and endpoint 0x81 for interrupt transfers.

A confusing aspect of interface descriptors is that the descriptor may have two IDs – an interface number and an alternative setting ID. The alternative setting value is a way of grouping interface descriptors together, so that one group can be switched for another group by simply changing the alternative setting. Since most USB toys only have a single interface descriptors, the alternative setting ID can be ignored (i.e. be set to 0 or -1).

Knowing a device's class can be useful for programming since many device classes have higher-level APIs built on top of the four basic transfer types: control, bulk, interrupt, and isochronous. Most USB toys belong to the HID class, which supports six higher-level requests types for transferring data (called *reports* in the HID documentation). The Windows HID API (details at http://msdn.microsoft.com/en-us/library/ms793246.aspx) provides an extensive set of functions for building, sending, and receiving reports. The bad news is that there isn't a Java binding for this API. That's quite a major hole in Java's USB support, bearing in mind the popularity of the HID class. However, it's fairly easy to drop back to using lower-level control transfers to implement HID-like functionality.

The "USB Made Simple" site at http://www.usbmadesimple.co.uk offers a good introduction to USB. A wonderful source for more information is Jan Axelson's website, http://www.lvr.com. He's also the author of the definitive book on USB:

> *USB Complete: The Developer's Guide*
> by Jan Axelson
> Lakeview Research; 4 ed., June 2009

"USB in a NutShell" at http://www.beyondlogic.org/usbnutshell/usb1.shtml is another simple online introduction. The USB specification is very readable, and can be found at http://www.usb.org/developers/docs/

For details on HID, see http://www.usb.org/developers/hidpage/ and http://www.lvr.com/hidpage.htm.

4. Becoming a USB Detective

Before I can start programming the launcher, I need to collect information about the device, including its vendor and product IDs, and the addresses of its endpoints. I also need to discover what forms of data transfer it uses.

The Dream Cheeky launcher comes with no technical information, so the easiest way to find its IDs is with a USB analysis tool, such as USBDeview (free from http://www.nirsoft.net/utils/usb_devices_view.html). Figure 6 shows USBDeview's brief list of *active* USB devices attached to my test machine.

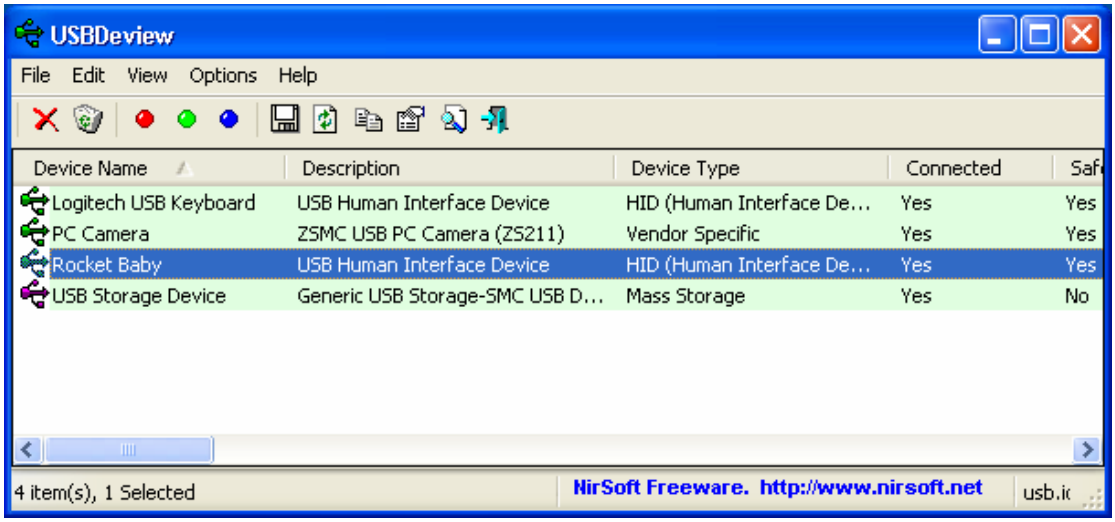

Figure 6. USBDeview's List of Active USB Devices.

Make sure that USBDeview is configured to exclude disconnected devices, or you'll receive a list of every USB device that's ever been connected to your PC!

Figure 6 shows four devices and, by a process of elimination (not really) reminiscent of Sherlock Holmes, I deduced that the Dream Cheeky launcher is identified as "Rocket Baby", a name which doesn't appear anywhere in its user documentation. The launcher's webcam is listed separately as "PC Camera".

Double clicking on the Rocket Baby row brings up more details, shown in Figure 7.

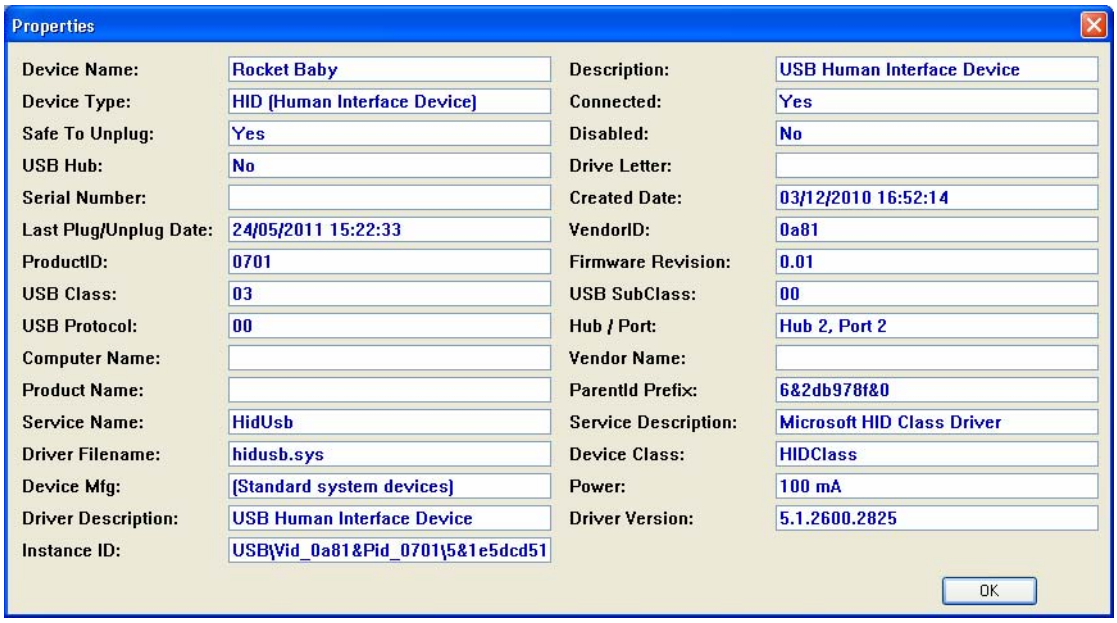

Figure 7. USBDeview's Details on Rocket Baby.

The important entries for my needs are the VendorID and ProductID hexadecimals: 0a81 and 0701 (about a third of the way down the two columns). Also, the device class is 0x03, which means it's a HID, but I won't be programming with the Windows HID API because there's no Java wrapper for it.

4.1. Analyzing the Dream Cheeky Launcher Protocol

I need to know what should go in the messages that go from the PC to the launcher. It's time to fire up a USB protocol analyzer.

I decided to use SysNucleus' USBTrace (http://www.sysnucleus.com/), a simple to use, powerful analyzer and packet filter that records USB I/O requests and other events. It's not free, but a 15-day trial period is a good way to try out the product. It requires the user to register for a free activation key.

In the past, I've used a similar tool called SourceUSB (http://www.sourcequest.com/), which has a 30-day trial period. Another popular, and free, analyzer is SnoopyPro from http://sourceforge.net/projects/usbsnoop/.

The first step with USBTrace is to select the device that should be monitored. Search through its device view tree (see Figure 8) for a gadget with the correct VendorID and ProductID hexadecimals (0a81 and 0701 for the launcher). Clicking on a device will make its details appear in an "Info" tab at the bottom-left of the display, where the IDs are 'hiding' in the "Hardware ID", USBVid_**0a81**&Pid_**0701**& Rev_0001.

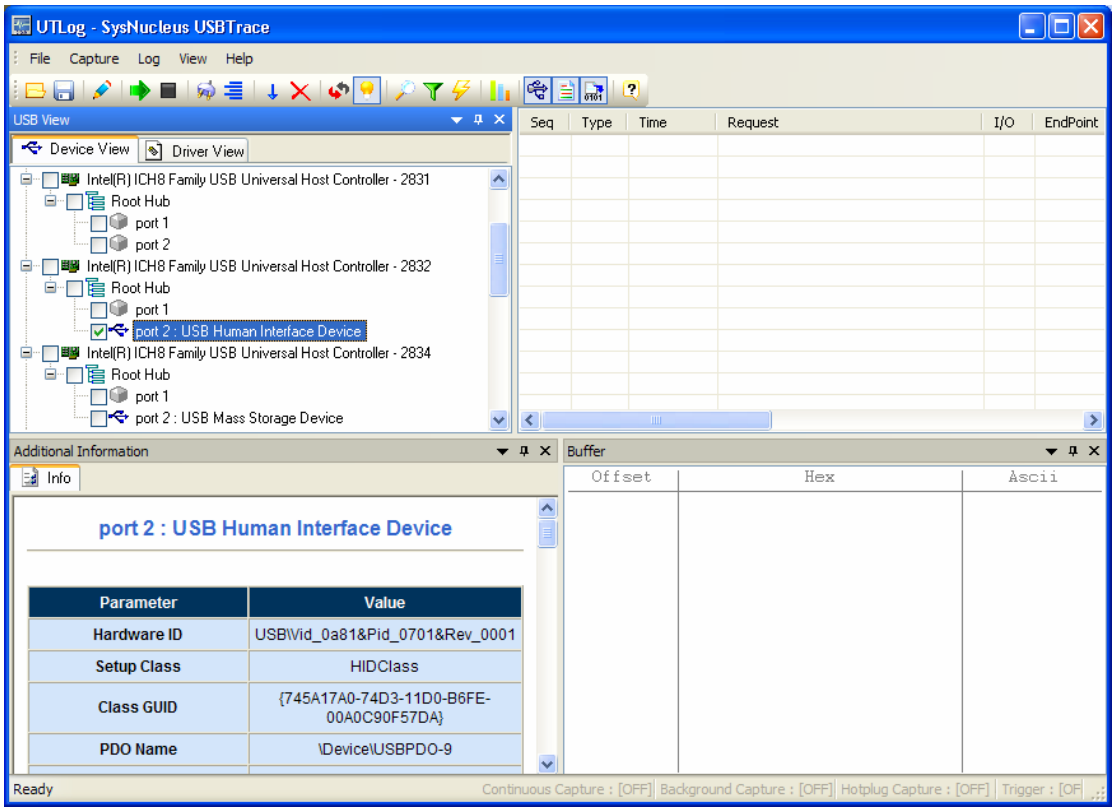

Figure 8. USBTrace Device Selection.

The "Info" tab contains a wealth (perhaps over-abundance) of information. Scrolling down shows the configuration, interface, and endpoint descriptors (see Figure 9).

Parameter	Value
Configuration Descriptor	
bLength	0x9
bDescriptorType	USB_CONFIGURATION_DESCRIPTOR_TYPE
wTotalLength	0x22
bNumInterfaces	0x1
iConfiguration	0x0
bmAttributes	0xA0 (Bus_Powered Remote_Wakeup)
MaxPower	0x32
Interface Descriptor	
bLength	0x9
bInterfaceNumber	0x0
bAlternateSetting	0x0
bNumEndpoints	0x1
bInterfaceClass	0x3 (Human Interface Device)
bInterfaceSubClass	0x0 (No Subclass)
bInterfaceProtocol	0x0 (None)
iInterface	0x0
Endpoint Descriptor	
bLength	0x7
bEndpointAddress	0x81 [IN]
bmAttributes	0x3 (USB_ENDPOINT_TYPE_INTERRUPT)
wMaxPacketSize	0x1
bInterval	0x14

Figure 9. USBTrace Descriptor Information.

The relevant things to note for later are that there's only a single configuration descriptor and interface descriptor, but there are *two* endpoints. The default 0 endpoint isn't listed, but it's joined by the 0x81 IN endpoint which receives interrupts coming from the device.

4.2. Rotating the Launcher

The next step is to turn on packet capturing for the device, and see what data is transmitted when I interact with the Dream Cheeky GUI (as shown in Figure 2). Figure 10 shows USBTrace's captured packets when I very briefly press the left rotation arrow.

Seq	Type	Time	Request	I/O	EndPoint	Device Object	IRP	Status	Buffer Snippet
0	START	0.000000	START OF LOG						
1	URB	5.701533	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EA7AA8	STATUS_SUCCESS	40
2	HID	5.701537	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EA7AA8	STATUS_SUCCESS	40
3	HID	5.701567	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EA7AA8	STATUS_PENDING	
4	HID	5.704847	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EA7AA8	STATUS_SUCCESS	
5	URB	5.706792	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	STATUS_SUCCESS	40
6	HID	5.706794	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	40
7	HID	5.706813	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_PENDING	
8	HID	5.709843	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	
9	URB	5.709871	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	STATUS_SUCCESS	04
10	HID	5.709873	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	04
11	HID	5.709890	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_PENDING	
12	HID	5.713842	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	
13	URB	5.814102	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	STATUS_SUCCESS	40
14	HID	5.814105	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	40
15	HID	5.814130	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_PENDING	
16	HID	5.817842	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	
17	URB	5.826844	BULK_OR_INTERRUPT_TRANSFER	OUT	81	\Device\USBPDO-9	0x85168008	STATUS_NOT_SUPPORTED	
18	URB	5.826847	BULK_OR_INTERRUPT_TRANSFER	OUT	81	\Device_HID00000005	0x85168008	STATUS_NOT_SUPPORTED	
19	URB	5.826867	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_PENDING	
20	URB	5.917615	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	STATUS_SUCCESS	40
21	HID	5.917618	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	40
22	HID	5.917640	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_PENDING	
23	HID	5.920839	(SetReport) CONTROL_TRANSFER	IN	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	
24	URB	5.938843	BULK_OR_INTERRUPT_TRANSFER	OUT	81	\Device\USBPDO-9	0x85175120	STATUS_NOT_SUPPORTED	
25	URB	5.938846	BULK_OR_INTERRUPT_TRANSFER	OUT	81	\Device_HID00000005	0x85175120	STATUS_NOT_SUPPORTED	
26	URB	5.938869	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_PENDING	
27	URB	5.938886	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	STATUS_SUCCESS	40
28	HID	5.938888	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	STATUS_SUCCESS	40

Figure 10. Packets Caught by USBTrace for a Left Rotation.

The volume of data is daunting, but USBTrace allows the log to be filtered in various ways. If I restrict it to only show successfully processed OUT messages (those going from the PC to the launcher), then the output becomes a little bit more manageable (see Figure 11).

Seq	Type	Time	Request	I/O	EndPoint	Device Object	IRP	Buffer Snippet	Buffer Size
0	START	0.000000	START OF LOG						
1	URB	5.701533	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EA7AA8	40	1
2	HID	5.701537	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EA7AA8	40	1
5	URB	5.706792	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	40	1
6	HID	5.706794	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	40	1
9	URB	5.709871	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	04	1
10	HID	5.709873	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	04	1
13	URB	5.814102	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	40	1
14	HID	5.814105	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	40	1
20	URB	5.917615	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	40	1
21	HID	5.917618	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	40	1
27	URB	5.938886	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EB47D0	40	1
28	HID	5.938888	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EB47D0	40	1
35	URB	6.042751	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x8507F0D8	40	1
36	HID	6.042755	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x8507F0D8	40	1
43	URB	6.052368	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84FC2B08	20	1
44	HID	6.052370	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84FC2B08	20	1

Figure 11. Successful OUT Packets Caught by USBTrace for a Left Rotation.

Each output message is shown twice – once as a HID SetReport CLASS INTERFACE message, and once as a lower-level control transfer CLASS INTERFACE message. This is indicated by the "Type" column which is either HID or URB (USB Request Block). Unfortunately, I can't filter by type to remove the HID messages, but let's ignore them anyway.

In this case, Figure 11 shows a series of CLASS INTERFACE messages being sent from the host (the PC) to the device (the launcher). To differentiate between them it's necessary to look at the messages' contents, which are shown in the Buffer panel at the bottom right of USBTrace, and in the Buffet Snippet and Buffer Size columns of the packets panel in Figure 11. Writing these message contents out as a sequence gives:

40, 40, **04**, 40, 40, …, 40, **20** // left turn

I've marked two 'clues' in bold. Remember that these messages were produced when the launcher was rotated *left*.

If I now reset the tracer and launcher, and briefly press the *right* arrow key, the series of CLASS INTERFACE messages produced this time (using the same filter) is shown in Figure 12.

Seq	Type	Time	Request	I/O	EndPoint	Device Object	IRP	Buffer Snippet	Buffer Size
0	START	0.000000	START OF LOG						
1	URB	1.905441	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EFA658	40	1
2	HID	1.905445	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EFA658	40	1
3	URB	1.912451	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EFA658	40	1
4	HID	1.912453	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EFA658	40	1
5	URB	1.922592	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84EFA658	08	1
6	HID	1.922594	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84EFA658	08	1
7	URB	2.026688	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84E60008	40	1
8	HID	2.026691	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84E60008	40	1
9	URB	2.131179	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x8505F3F0	40	1
10	HID	2.131182	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x8505F3F0	40	1
11	URB	2.147773	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x84FA9008	20	1
12	HID	2.147776	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x84FA9008	20	1

Figure 12. Successful OUT Packets Caught by USBTrace for a Right Rotation.

If I again extract the message contents as a sequence, then I see:

40, 40, **08**, 40, 40, …, 40, **20** // right turn

If I repeat this for up and down rotations, the sequences are:

40, 40, **02**, 40, 40, …, 40, **20** // upwards

40, 40, **01**, 40, 40, …, 40, **20** // downwards

Based on **when** these message are generated, the 0x04, 0x08, 0x02, and 0x01 messages start a rotation, while 0x20 stops it (when the user releases the arrow key in the GUI).

What about all the 0x40s messages? To be honest, I'm not sure what they do. They may be polling or "stay alive" messages; I'll come back to them when I start programming. The results of my investigations are summarized in Figure 13.

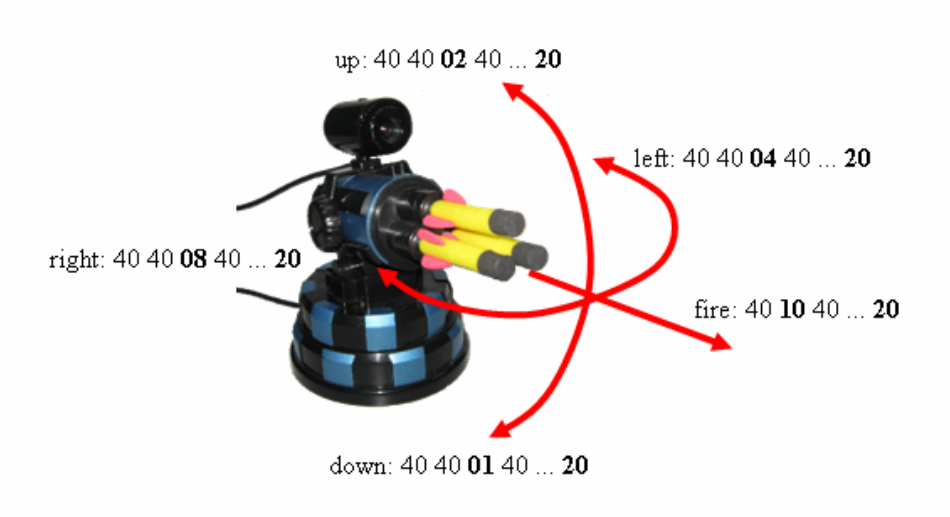

Figure 13. Control Transfer Messages for the Launcher.

4.3. Firing a Missile

The GUI interface for firing a missile is the red button in the center of the star shape in Figure 2. Once it's pressed, air pressure builds up behind a missile until it's shot from the launcher. Figure 14 shows the successful OUT packets sent to the host during that time.

Seq	Type	Time	Request	I/O	EndPoint	Device Object	IRP	Buffer Snippet	Buffer Size
0	START	0.000000	START OF LOG						
1	URB	1.952456	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x85166CE8	40	1
2	HID	1.952459	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x85166CE8	40	1
3	URB	1.955667	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x85166CE8	10	1
4	HID	1.955670	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x85166CE8	10	1
5	URB	1.959636	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x85166CE8	40	1
6	HID	1.959639	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x85166CE8	40	1
7	URB	2.063374	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x851A0608	40	1
8	HID	2.063377	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x851A0608	40	1
9	URB	2.082631	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x851A0608	40	1
10	HID	2.082633	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x851A0608	40	1
11	URB	2.186419	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x851A0608	40	1
12	HID	2.186423	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x851A0608	40	1
			// many 0x40 messages						
1...	URB	7.546721	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x85166CE8	40	1
1...	HID	7.546724	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x85166CE8	40	1
1...	URB	7.570511	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x85166CE8	40	1
1...	HID	7.570513	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x85166CE8	40	1
1...	URB	7.674494	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x85166CE8	40	1
2...	HID	7.674497	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x85166CE8	40	1
2...	URB	9.905860	CLASS_INTERFACE	OUT	0	\Device\USBPDO-9	0x85089568	20	1
2...	HID	9.905863	(SetReport) CLASS_INTERFACE	OUT	0	\Device_HID00000005	0x85089568	20	1

Figure 14. Successful OUT Packets Caught by USBTrace for a Missile Fire.

If I again extract the message contents as a sequence, then I get:

40, **10**, 40, 40, …, 40, **20** // fire

The 0x10 message initiates the build up of pressure, the missile is ejected 5-7 seconds later, and the 0x20 message stops the pressure. The time required for a missile to fire

can vary quite a lot, and sometimes there seems to be pressure left over from the previous missile firing which lets the current missile be released faster.

4.4. Interrupts

So far I've been focusing on control transfers which go via endpoint 0. The launcher uses these messages to start rotating, start a firing sequence, and to stop. However, the launcher also sends out interrupts, arriving at the PC via endpoint 0x81. Let's return to USBTrace, to decide how these interrupts are used.

Interrupts for Rotation Limits

If a user holds down the left, right, up, or down arrow on the GUI, then the launcher keeps turning until it gets to an extreme position, then stops by itself. Even if the user keeps pressing the arrow, the launcher won't turn any further.

This mechanism is implemented using interrupts, which can be seen by rotating the launcher to its left-most limit. Figure 15 shows that interrupts containing the message 0x00 are periodically sent to the host while the rotation is okay, but 0x04 is delivered in the last row of the figure when the limit is reached. Figure 15 only lists incoming messages with data arriving through endpoint 0x81.

Seq	Type	Time	Request	I/O	EndPoint	Device Object	IRP	Status	Buffer Snippet	Buffer Size
77	URB	5.147050	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	00	1
85	URB	5.163048	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_SUCCESS	00	1
93	URB	5.275190	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	00	1
1...	URB	5.291047	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_SUCCESS	00	1
1...	URB	5.387045	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	00	1
1...	URB	5.403045	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_SUCCESS	00	1
1...	URB	5.499043	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	00	1
1...	URB	5.515042	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_SUCCESS	00	1
1...	URB	5.627040	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	00	1
1...	URB	5.643039	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_SUCCESS	00	1
1...	URB	5.739038	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	00	1
1...	URB	5.755038	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_SUCCESS	00	1
1...	URB	5.851036	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	00	1
1...	URB	5.867035	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85175120	STATUS_SUCCESS	00	1
1...	URB	5.963032	BULK_OR_INTERRUPT_TRANSFER	IN	81	\Device_HID00000005	0x85168008	STATUS_SUCCESS	04	1

Figure 15. Successful IN Packets via Endpoint 0x81 for a Left Rotation.

For a right rotation, interrupts are used in a similar way: 0x00 means okay but an interrupt containing the message 0x08 signals that the limit has been reached. For a down rotation, the limit message is 0x01, and for upwards turning the message contains 0x02

Another aspect of these interrupts is that several limit interrupts may be sent, even after the launcher has rotated away from an extreme position.

Firing Interrupts

Interrupts are also utilized in missile firing. A sequence of 0x00 interrupt messages are sent to the device during pressure build-up, but a 0x10 message when the missile has left the launcher. Subsequently, the host sends a 0x20 control message to stop the

firing sequence. Figure 16 adds these interrupt values to the control transfer information of Figure 13.

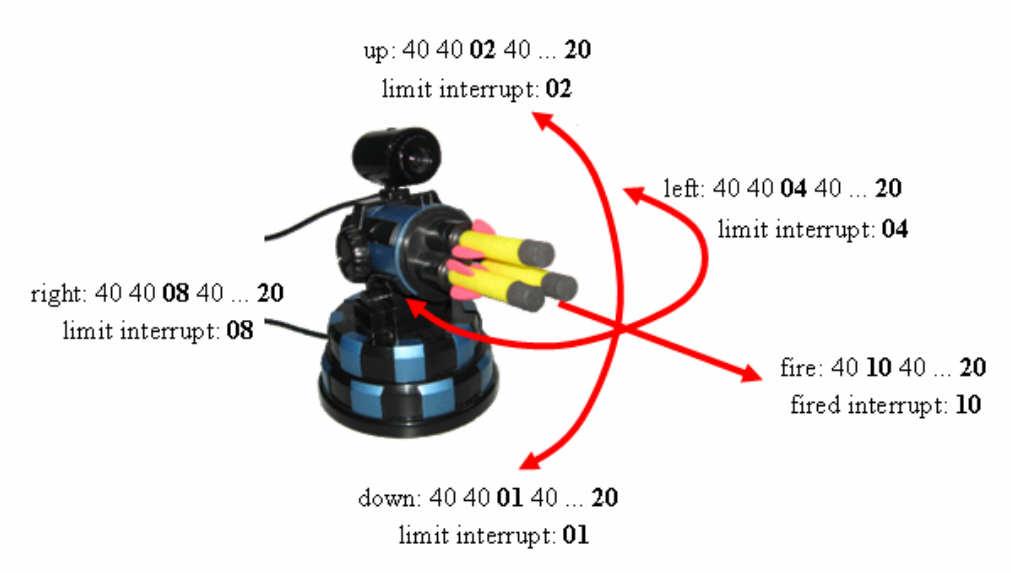

Figure 16. Control Transfer and Interrupt Messages for the Launcher.

5. Installing USB Support for Java

There's an official USB extension for Java, called JSR-80, located at http://javax-usb.org. However, the Windows implementation has been in an alpha state for several years, and I had problems getting it to work. Instead I went for LibusbJava (http://libusbjava.sourceforge.net/wp/) which requires me to install three main elements, as shown in Figure 17.

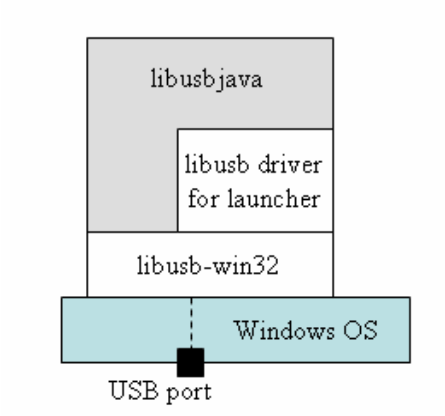

Figure 17. USB Support for Java in Windows.

5.1. Installing libusb-win32

LibusbJava relies on libusb-win32, a Windows port of a widely used USB library for Linux. The current version of libusb-win32 can be downloaded from http://sourceforge.net/apps/trac/libusb-win32/wiki (I retrieved libusb-win32-bin-1.2.4.0.zip).

The executable installs libusb-win32 into c:\Program Files\LibUSB-Win32\. Inside its bin\ subdirectory are DLLs for various Windows architectures which have to be renamed and moved over to the OS. I used the x86\ DLLs, moving libusb0_x86.dll to Windows\system32\libusb0.dll (notice the rename), and libusb0.sys to Windows\systems32\drivers\libusb0.sys (notice the unchanged name).

testlibusb-win.exe, located in x86\, can be employed to test the installation by listing out the USB devices using libusb-win32 drivers. That's not much use yet since I don't have any libusb drivers installed..

The linusb-win32 documentation is a bit sketchy, but lots more can be found at the website (http://sourceforge.net/apps/trac/libusb-win32/wiki). Also accessible from there is API documentation, a developers guide, a FAQ, and a mailing list.

5.2. A libusb-win32 driver for the Launcher

The libusb-win32 bin\ directory contains the inf-wizard.exe tool, which can create a INF file for a libusb-win32 driver. inf-wizard.exe starts by listing all the devices connected to the PC, listed by their vendor ID, product ID, and device description (see Figure 18).

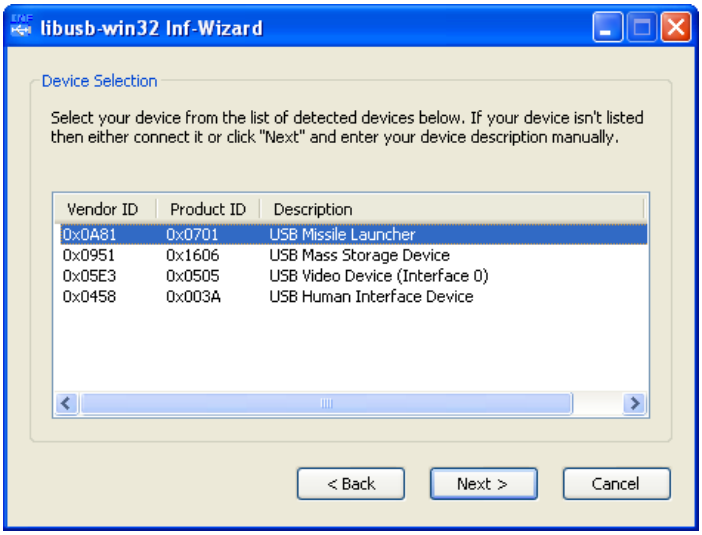

Figure 18. The inf-wizard.exe Application.

I know which entry to choose by referring to the vendor and product IDs I obtained from USBDeview (0A81 and 0701; see Figure 7). After my selection, inf-wizard.exe generates an INF file for the launcher, which I can install in the OS by right clicking on it.

Now I can use testlibusb-win.exe to check the launcher details, as shown in Figure 19.

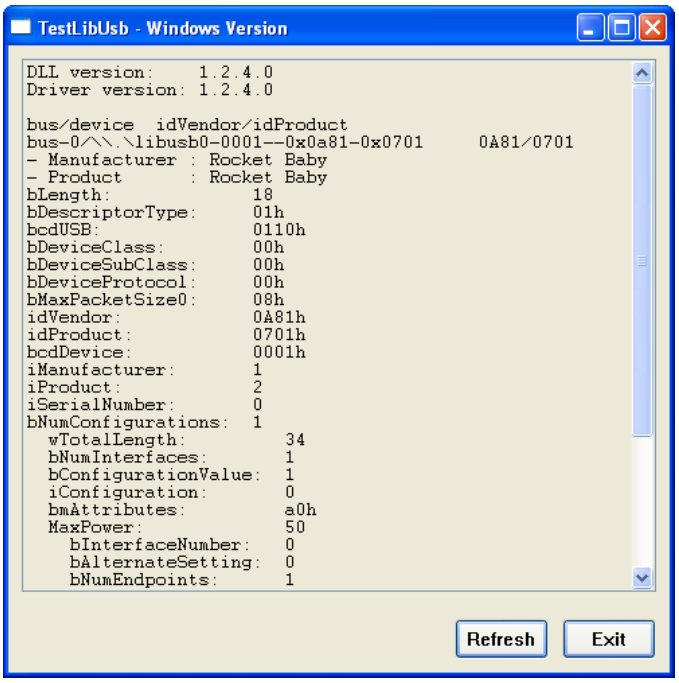

Figure 19. The testlibusb-win Details for The Launcher.

I can also examine the device via Windows Device manager, which is reached via the Hardware tab of the System control panel.

The launcher has two USB components – the launcher base which now has a libusb-win32 driver, and the camera stuck on top of the launcher. The webcam doesn't require a driver since I'll utilize JMF to communicate with the driver installed by the Dream Cheeky AMCAP camera viewing application. I'll explain the details later.

5.3. Installing LibusbJava

Once libusb-win32 is installed, the LibusbJava library can be downloaded from http://libusbjava.sourceforge.net/wp/. The necessary files are a JAR (ch.ntb.usb-0.5.9.jar) and a zipped DLL (LibusbJava_dll_0.2.4.0.zip). I placed the JAR and unzipped DLL in a directory on my c:\ drive (c:\libusbjava\), but anywhere is fine.

The JAR includes a number of test applications, the simplest being a viewer for the libusb-win32 USB devices connected to the machine. Figure 20 shows its output for the Dream Cheeky launcher.

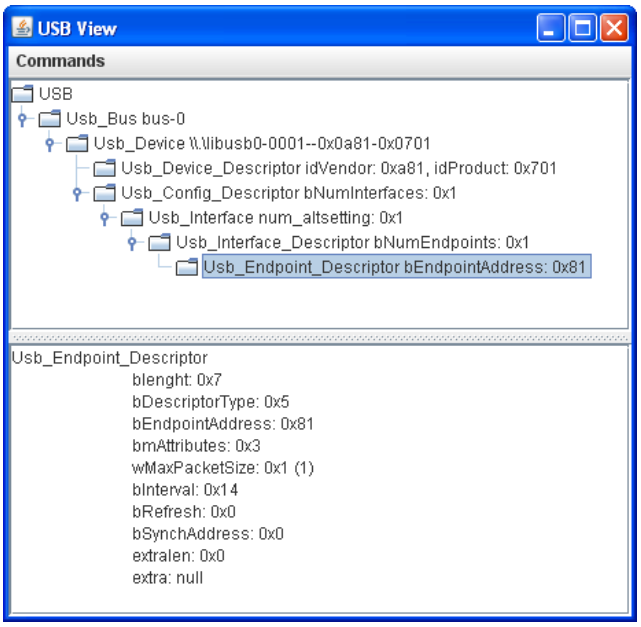

Figure 20 The USB View Application.

Note that this application only displays information on libusb-win32 drivers.

The viewer (which is inside ch.ntb.usb-0.5.9.jar) is started using the command line:

```
java -Djava.library.path="c:\libusbjava"
     -cp "c:\libusbjava\ch.ntb.usb-0.5.9.jar;."
                                  ch.ntb.usb.usbView.UsbView
```

The paths to the DLL and JAR files need to match their location on your machine.

6. Using the LibusbJava Library

LibusbJava offers two slightly different ways of carrying out USB programming. Its ch.ntb.usb.LibusbJava class is a thin layer over libusb-win32 so it's quite easy to translate C/C++ code using libusb-win32 into Java. The API also offers a slightly more high-level interface, which mostly differs in the way that USB devices are initialized. For example, the ch.ntb.usb.Device class makes it easier to handle errors and timeouts. I'll be using this latter approach for my coding.

The LibusbJava website includes two examples illustrating these coding styles, which are also explained in the API documentation at http://libusbjava.sourceforge.net/wp/res/doc/. There's a useful forum at http://sourceforge.net/projects/libusbjava/forums/forum/660151/index/

Initializing the Launcher (and Closing Down)

The great advantage of using the LibusbJava higher-level classes is the simplicity of opening and closing a device. Most programs have the following structure:


```
public static void main(String[] args)
{
  Device dev = USB.getDevice(VENDOR_ID, PRODUCT_ID);
  try {
    dev.open(1, 0, -1);
    // open device with configuration 1, interface 0
    // and no alternative interface

    // communicate with the device (see below for details)

    dev.close();    // close down
  }
  catch (USBException e) {
    System.out.println(e);
  }
}  // end of main()
```

The vendor and product IDs are the hexadecimal IDs which we've seen many times (e.g. 0x0a81 and 0x0701 in Figure 7).

The first argument of Device.open() is the configuration value, the second is the interface number, and the last is the alternate interface, which can be set to 0 or -1 if there isn't one.

The configuration value and interface number can be obtained from USBTrace Descriptor Information, as shown in Figure 9. The configuration value is the table entry called iConfiguration and the interface number is in the bInterfaceNumber row. For USB toys, the values are almost always 1 and 0.

6.1. Making the Launcher Move

Figure 16 summarizes what data has to be sent between the PC and the launcher, which falls into two categories. I need to send control transfer messages to the launcher to start it moving (e.g. 0x04 will rotate it left). To stop the launcher, I must send a 0x20 message.

You may be wondering about all the 0x40 messages. I discovered through experimentation, that these aren't needed to start and stop the launcher, or to trigger missile firing. Due to their position in the USB communications log just before interrupt messages, my bet is that they're status request messages. The arrival of a 0x40 message at the launcher triggers the gathering of data that's sent back in the subsequent interrupt read. Assuming this, I can simplify Figure 16 to become Figure 21.

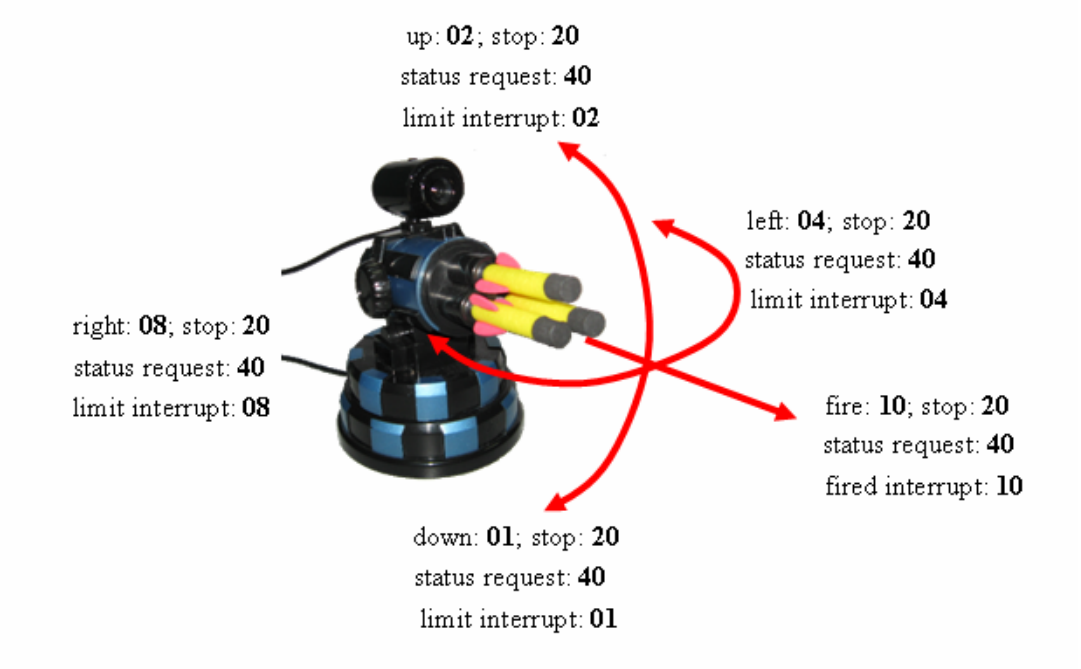

Figure 21. Simplified Control Transfer and Interrupt Messages for the Launcher.

In the Device class, the prototype for sending a control transfer is:

```
public int controlMsg(int requestType,
                      int request, int value, int index,
                      byte[] data, int size,
                      int timeout,boolean reopenOnTimeout)
               throws USBException
```

There's a brief explanation of the method in the libusbJava API documentation at http://libusbjava.sourceforge.net/wp/res/doc/, which refers to the USB control transfer specification in sections 9.3 and 9.4 of the Revision 2.0 document (available at http://www.usb.org/developers/docs/).

The method's parameters use (almost) the same names as the USB control transfer explained in the specification, and returns the number of bytes written (or a negative number if there's an error).

In practice, the best way of knowing what arguments should be passed to Device.controlMsg() is by looking at the packet information displayed by USBTrace when the launcher GUI transmits a control transfer.

For example, Figure 22 shows packet details for the control transfer that sends 0x04 to the device, to start it turning left.

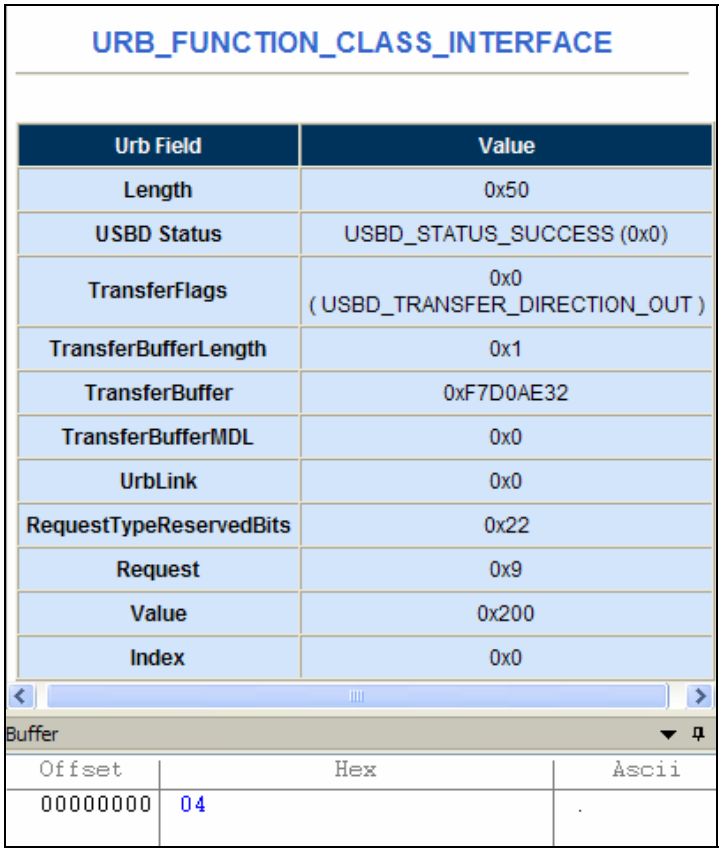

Figure 22. Control Transfer for Moving Left.

Figure 22 shows values that can be used as Device.controlMsg() arguments for the request, value, index, and size of the byte array (TransferBufferLength). Also, the method's data argument should be a byte array containing only 0x04 (the Buffer contents in Figure 22).

Device.controlMsg()'s requestType argument appears as the RequestTypeReservedBits value (0x22) in the table. The bits are a mix of three settings: the data transfer direction, the device type, and the recipient type. Based on a study of LibusbJava's documentation for constants, 0x22 is a combination of USB.REQ_TYPE_DIR_HOST_TO_DEVICE, USB.REQ_TYPE_TYPE_CLASS, and USB.REQ_TYPE_RECIP_ENDPOINT.

To my dismay, this collection of bits did **not** work when I tried them in Device.controlMsg(). I was forced to try other bit combinations until I arrived at USB.REQ_TYPE_DIR_HOST_TO_DEVICE, USB.REQ_TYPE_TYPE_CLASS, and USB.REQ_TYPE_RECIP_**INTERFACE** (which is equivalent to 0x21). In other words, the transfer must be sent to an interface rather than an endpoint. The difference is that an interface is a collection of endpoints, while an endpoint is a single device port. To be fair, this bit of guesswork wasn't too hard, since it corresponds to the name of the USB function, CLASS_INTERFACE, shown at the top of Figure 22.

Device.controlMsg()'s timeout value isn't part of the USB specification, and is used by libusb to decide how long to wait for a response before signaling an error. The method's reopenOnTimeout argument is a LibusbJava boolean that attempts to retry the operation after a timeout when set to true.

My sendControl() function uses Device.controlMsg() to build and send a control transfer over to the device.

```
private static void sendControl(Device dev, int opCode)
{
  System.out.println("Sending opCode: " + toHexString(opCode));
  byte[] bytes = { new Integer(opCode).byteValue() };
  try {
    int rval = dev.controlMsg(
                          USB.REQ_TYPE_DIR_HOST_TO_DEVICE |
                          USB.REQ_TYPE_TYPE_CLASS |
                          USB.REQ_TYPE_RECIP_INTERFACE,
                  0x09, 0x0200, 0,
                  bytes, bytes.length, 2000, false);
    if (rval < 0) {
      System.out.println("Control Error (" + rval + "):\n  " +
                          LibusbJava.usb_strerror() );
    }
  }
  catch (USBException e) {
    System.out.println(e);
  }
}  // end of sendControl()


private static String toHexString(int b)
// change the hexadecimal integer into "0x.." string format
{
  String hex = Integer.toHexString(b);
  if (hex.length() == 1)
    return "0x0" + hex;
  else
    return "0x" + hex;
}  // end of toHexString
```

sendControl() can transmit any control transfer, such as 0x04 to start the launcher turning left. Of course, I also want to stop the rotation after a certain amount of time, and that can be implemented by sending 0x20 after waiting for the required period.

The following main() function makes the launcher rotate to the left for 2 seconds.

```
public static void main(String[] args)
{
  Device dev = USB.getDevice((short)0x0a81, (short)0x0701);
  try {
    dev.open(1, 0, -1);

    sendControl(dev, 0x04);   // start launcher rotating left
    wait(2000);               // wait 2 secs while launcher is moving
    sendControl(dev, 0x20);   // stop launcher

    dev.close();
  }
  catch (USBException e) {
    System.out.println(e);
  }
}  // end of main()
```



```
private static void wait(int ms)
// sleep for the specified no. of millisecs
{ try {
    Thread.sleep(ms);
  }
  catch(InterruptedException e) {}
}  // end of wait()
```

6.2. USB Devices are Temperamental

The main() function for rotating the launcher is nicely brief, and easy to understand. However, it doesn't convey the headaches that can occur when programming USB gadgets. Here are some techniques I've developed for avoiding, or at least reducing, the problems.

It *really* helps to have *two* test machines. In the case of the Dream Cheeky launcher I installed its GUI on one machine, and the LibusbJava and libusb-win32 libraries, and the libusb-win32 generated driver on another. In this way there's no chance of having two device drivers for the same USB gadget interfere with each other. At the programming level, this kind of interaction problem often reveals itself as the informative Windows USB error "a device attached to the system is not functioning".

If a device driver generates an error, the only certain way to correctly reset it is to reboot the machine. This can add many hours to the development time.

Although USBTrace is a great tool, it may not report the full story of what's happening at the OS kernel and hardware levels. For instance, my problem with the control transfer's request type is probably due to the kernel converting the type from an endpoint to the device's interface.

Always write a simple test-rig (e.g. like the code above) to test the communication protocol before moving to a more complex application. Small headaches are easier to cure than big ones.

6.3. Status Monitoring

The other part of programming the launcher is status monitoring, which comes in two forms – checking if the launcher has reached a rotation limit, and deciding whether a missile has been shot from the launcher.

As Figure 21 suggests, status monitoring is initiated by sending a 0x40 to the device followed by an interrupt read to get back the gathered data. If a rotation limit hasn't been reached, or a missile hasn't yet left the launcher then 0x00 is returned, or sometimes a timeout. There are four different rotation limit messages, 0x01, 0x02, 0x04, and 0x08, which correspond to the opcodes used to start that type of rotation. Similarly, data signaling a missile release is 0x10, the same opcode that requests the firing.

Before I go any further, I should warn delicate readers that a nasty bit of hacking lies ahead. Despite many hours of experimentation I was unable to get status monitoring to work via LibusbJava; the best I could achieve were interrupt reads that always timed out, without returning any data. Nevertheless, I'll explain my approach since it

should be correct, and it may help readers developing code for other USB gadgets, or a more recent version of the Dream Cheeky launcher.

If anyone reading this chapter solves my problem, I'd be very happy to update the code at the NUI website (http://fivedots.coe.psu.ac.th/~ad/jg/??) and in this chapter, giving you due thanks for your efforts.

But just a cotton-pickin' moment, what about Figure 1 which shows the author suffering before the frightening onslaught of the mighty launcher. Is the MotionLauncher application working or not? It's working, but instead of using status monitoring with interrupts to detect rotation limits and successful firing, I use time estimates. I'll explain the ugly details presently, after talking about interrupts.

The LibusbJava method I need for reading data from interrupts is Device.readInterrupt(), which has the prototype:

```
public int readInterrupt(int in_ep_address,
                   byte[] data, int size,
                   int timeout, boolean reopenOnTimeout)
                 throws USBException
```

The arguments are similar to those for Device.controlMsg(), except for in_ep_address which is the endpoint to read from. This information is easy to find by looking at the USB descriptor details in USBTrace, as shown in Figure 23.

Endpoint Descriptor	
bLength	0x7
bEndpointAddress	0x81 [IN]
bmAttributes	0x3 (USB_ENDPOINT_TYPE_INTERRUPT)
wMaxPacketSize	0x1
bInterval	0x14

Figure 23. USB Descriptor Information for Endpoints.

According to Figure 23, the endpoint address is 0x81, and the byte[] array needs to be big enough to hold 1 byte.

Another way of obtaining this information is by double clicking on an interrupt packet in USBTrace's log. For example, Figure 24 shows details about a 0x01 interrupt transfer, issued when the launcher can't rotate downwards any further.

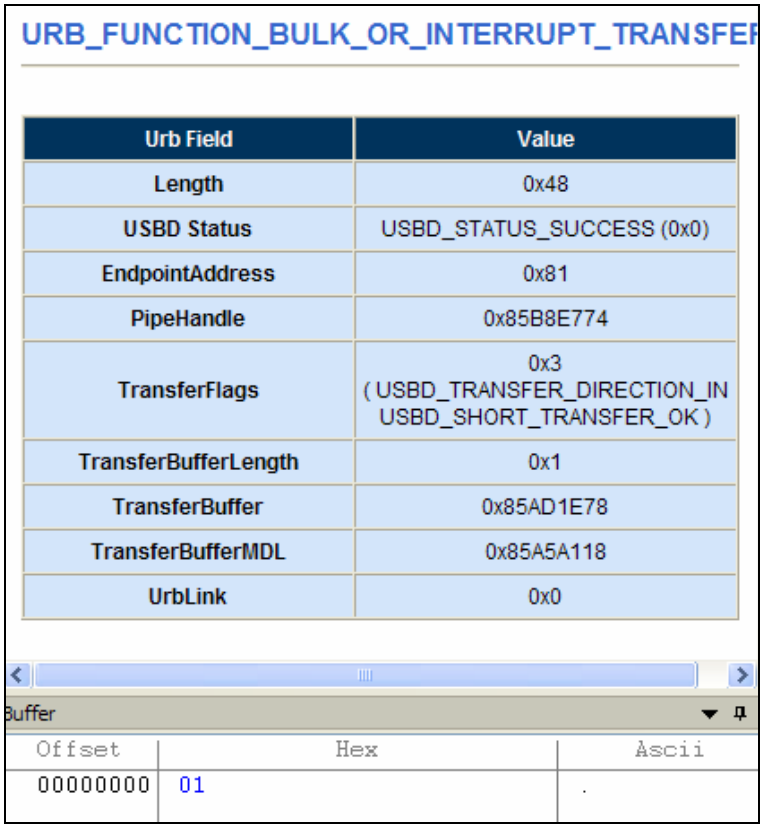

Figure 24. Interrupt Message for Downwards Limit.

My method for extracting status information from a Device.readInterrupt() call is:

```
private static int readStatus(Device dev, int timeout)
{
  byte[] buffer = new byte[1];
  try {
    int res = dev.readInterrupt(0x81, buffer, buffer.length,
                                              timeout, false);
    if (res < 0)
      System.out.println("Interrupt read error (" + res + "):\n " +
                                    LibusbJava.usb_strerror() );
    else if (res == 0)
      System.out.println("No data in interrupt read");
    else { // res > 0
     return ((int)buffer[0] & 0xff);      // return status data
  }
  catch (USBTimeoutException e) {
    System.out.println("interrupt timeout");
  }
  catch (USBException e) {
    System.out.println(e);
  }
  return -1;
}  // end of readStatus()
```

Error cases, such as Device.readInterrupt() timing out are reported as a negative result. Status data should be positive (0x00 or greater).

There needs to be a status request (i.e. a control transfer using 0x40) before the interrupt read. I combine this with timeout functionality in waitUntil():

```
// global
private static final int INTERRUPT_TIME_OUT = 100; // ms


private static boolean waitUntil(Device dev, int maxTime,
                                                    int intrCode)
/* wait until maxTime ms has passed, if not interrupted first;
   return after maxTime ms with a true result or earlier
   with a false result if the interrupt transfer returns
   the right interrupt code (intrCode)  */
{
  int totalWait = 0;
  while (totalWait < maxTime) {
    sendControl(dev, 0x40);    // send status request
    if (readStatus(dev, INTERRUPT_TIME_OUT) == intrCode)
      return false;
    totalWait += INTERRUPT_TIME_OUT;
  }
  return true;
}  // end of waitUntil()
```

waitUntil() waits for maxTime milliseconds to pass, but instead of sleeping with Thread.sleep(), it keeps requesting status information. Each request may take up to INTERRUPT_TIME_OUT milliseconds, which is the timeout for the Device.readInterrupt() call inside readStatus().

The status value returned by readStatus() must be compared with the interrupt opcode in intrCode to decide if the waiting should be terminated early.

waitUntil() returns true if the full maxTime time has passed, or false if the waiting has finished early because of a status report.

An example of how waitUntil() can be used with a downwards rotation:

```
sendControl(dev, 0x01);     // start launcher rotating down
boolean timeoutReached = waitUntil(dev, 3000, 0x01);
                            // wait 3 secs or until status is 0x01
sendControl(dev, 0x20);     // stop launcher
```

The downward rotation may last for a maximum of 3 seconds, but waitUntil() can finish earlier if the status report returns 0x01 (meaning that the downward limit has been reached). In either case, after waitUntil() returns, the rotation is stopped.

As I mentioned above, this code does **not** work correctly. Even when a rotation reaches its physical limit, the call to Dev.readInterrupt() in readStatus() only timeouts (which is caught by the USBTimeoutException catch block in readStatus()).

I tried numerous variations of this idea, such as issuing more than one 0x40 control transfer before an interrupt read, varying the time delays between the control transfer and the interrupt read. I tried moving the code to a separate Java thread, and have it repeatedly call waitUntil() with small max times until the device was closed. None of these changes made waitUntil() work correctly.

At the end of this chapter, there are links to other Dream Cheeky launcher applications and libraries on Windows, Mac, and Linux, using languages such as C, C#, Python, and Perl. It's clear that status monitoring has posed problems for other people. Some implementations don't mention monitoring at all.

6.4. Replacing Monitoring with Explicit Time Limits

Let the hacking commence: I'll replace status monitoring with time limits, summarized in Figure 25.

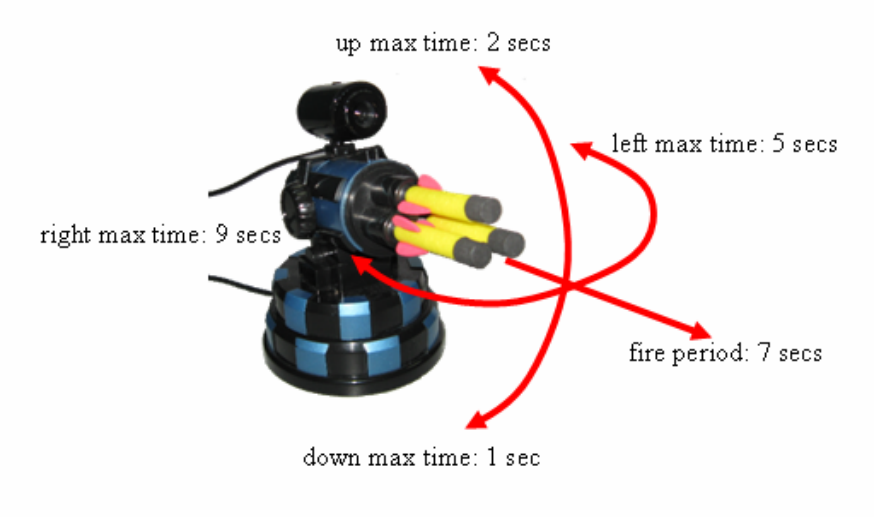

Figure 25. Rotation and Firing Time Limits.

I measured the time it took for the launcher to rotate from its default central position to its rotation limits in the left, right, up, and down directions. I also timed how long it took for the pump inside the launcher to build up sufficient pressure to expel a missile. I rounded down to the nearest integer for the rotation limits, and rounded up the fire period. I now have the maximum amount of time that the launcher can spend rotating horizontally and vertically.

During rotations I'll store the times spent moving horizontally and vertically. Both times start at 0 at the launcher's central start position. A movement to the left decrements the horizontal 'time', while a move to the right increments it. Similarly, a movement up increments the vertical time, while rotating downward decrements it. When these ongoing horizontal and vertical times reach the maximum time limits shown in Figure 25 then further rotations in that direction is disabled.

This approach requires me to know the duration of a rotation, but that's included in every rotation command as the wait time between sending a start rotation command and a stop.

The following code snippets shows the use of the resulting MissileLauncher class, the internals of which I'll explain shortly.

```
MissileLauncher launcher = new MissileLauncher();

launcher.up(700);      // move up for 700 ms
launcher.fire();
```



```
launcher.down(900);
launcher.down(1000);   // rejected since limit would be exceeded

launcher.left(1000);

launcher.reset();      // restores launcher to starting position

launcher.close();
```

The launcher will rotate upwards, fire a missile, then rotate below its starting position. It will not execute the second MissileLauncher.down() call since that would move the gun below its downward limit (1 second according to Figure 25). An error message is printed, and the call to down() returns false. The down limit doesn't stop the next rotation, which is to the left. The reset() call uses the stored on-going horizontal and vertical times to undo the rotations, restoring the launcher to its starting position.

6.5. Integration with the Motion Detector

The launcher is going to be integrated with my motion detection code, but there's a slight mismatch between the two. The motion detector (see the previous chapter) reports how a center-of-gravity (COG) has moved in terms of its pixel displacement from its previous position. I need to convert these pixel distances into time arguments suitable for MissileLauncher.up(), and the other rotation methods.

My solution is another time hack, based on how long it takes the launcher to rotate across one webcam image's width and height. For instance, the left hand side of Figure 26 shows a man standing on the left edge of the webcam image. I timed how long it took the launcher (and the webcam image) to rotate left until the man was positioned at the right of the image.

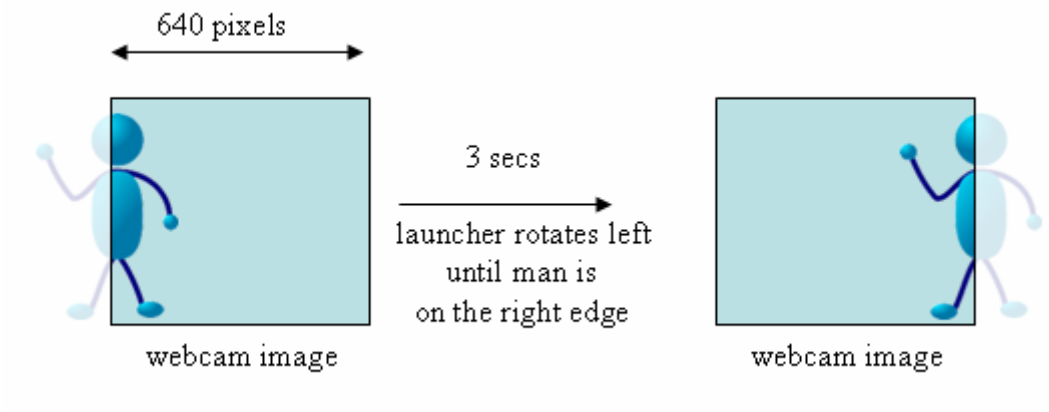

Figure 26. Timing the Horizontal Traversal of a Webcam Image,

This allows me to link image pixel distance and launcher rotation time, since I know the width of the webcam image (640 pixels) and the rotation time (3 seconds). I can say (roughly) that it takes 1 second for the launcher to traverse 213 pixels.

I carried out a similar measurement for the vertical traversal of an image. The launcher moves more slowly in that direction, taking about 3 seconds to cross 480 pixels.

This approach is a long way from being accurate. For example, I only timed things to the nearest second, and I'm assuming that the launcher's velocity is constant. This is obviously not true if you consider what happens when a single large rotation is replaced by several smaller ones. For instance, instead of calling MissileLauncher.left() once for 3 seconds, call it three times, each for 1 second. The total distance traveled in both cases is not the same (although it is close). Inaccuracies accumulate due to the mechanical starting and stopping of the launcher.

What this approach does have going for it is simplicity, both to explain and implement. After all, I'm not developing a launcher system that has to hit a pin on the moon; the inaccuracies are perfectly reasonable for tracking a person moving a few feet from the webcam.

7. Implementing the MissileLauncher Class

My MissileLauncher class simplifies the opening of a connection to the launcher, and offers high-level rotation and firing methods that hide control transfers and the opcodes that drive the launcher. The class contains hardwired timing values for rotation limits and the firing period (as shown in Figure 25).

The basic rotation methods are left(), right() up(), and down(), which all take a time period argument. There are also two rotation methods that take pixel arguments, moveHorizontal() and moveVertical(), which convert pixel distances into time periods and then call the relevant left(), right() up(), or down() functions.

The MissileLauncher constructor takes the dimensions of the webcam image as arguments, which are later used to convert pixel lengths to times. These only give correct answers if the dimensions are in the same ratio as a default image size (640 pixels wide / 480 deep). If the ratio is different, then the code issues a warning.

The launcher is opened using LibusbJava 's USB and Device classes.

```
// globals
// default image size
private final static int SCR_WIDTH = 640;
private final static int SCR_HEIGHT = 480;

private final static short VENDOR_ID = (short)0x0a81;
private final static short PRODUCT_ID = (short)0x0701;
       // IDs for the launcher

private Device dev = null;
              // used to communicate with the USB device

private int scrWidth, scrHeight;
       /* dimensions of the screen (I assume they're the
          same width/height ratio as SCR_WIDTH/SCR_HEIGHT) */


public MissileLauncher()
```



```
{  this(SCR_WIDTH, SCR_HEIGHT);  }


public MissileLauncher(int w, int h)
{
  scrWidth = w;
  scrHeight = h;

  // check image dimension ratio against defaults
  if ((scrWidth*SCR_HEIGHT) != (SCR_WIDTH*scrHeight))
     System.out.println("Screen size ratio doesn't
                                    match conversion defaults");

  System.out.println("Looking for device: (vendor: " +
                   toHexString(VENDOR_ID) +
                  "; product: " + toHexString(PRODUCT_ID) + ")");
  dev = USB.getDevice(VENDOR_ID, PRODUCT_ID);
  try {
    System.out.println("Opening device");
    dev.open(1, 0, -1);
    // configuration 1, interface 0 and no alt interface
  }
  catch (USBException e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of MissileLauncher()
```

7.1. Basic Rotation Methods

The basic rotation methods (left(), right() up(), and down()) all utilize sendCommand(), which makes two calls to sendControl(), separated by some waiting:

```
// globals
private Device dev = null;   // the USB device


private void sendCommand(int opCode, int period)
// execute the opCode operation for period millisecs
{
  if (dev != null) {
    sendControl(opCode);  // start the operation
    wait(period);
    sendControl(0x20);    // stop the operation
  }
}  // end of sendCommand()


private void sendControl(int opCode)
// send a USB control transfer
{
  System.out.println("Sending opCode: " + toHexString(opCode));
  byte[] bytes = { new Integer(opCode).byteValue() };

  try {
    int rval = dev.controlMsg(
                      USB.REQ_TYPE_DIR_HOST_TO_DEVICE |
                      USB.REQ_TYPE_TYPE_CLASS |
```



```
                    USB.REQ_TYPE_RECIP_INTERFACE,
                0x09, 0x0200, 0,
                bytes, bytes.length, 2000, false);
    if (rval < 0) {
      System.out.println("Control Error (" + rval + "):\n  " +
                                  LibusbJava.usb_strerror() );
    }
  }
  catch (USBException e) {
    System.out.println(e);
  }
}  // end of sendControl()
```

Each of the rotation function calls sendCommand() with the correct opCode, and also updates the ongoing horizontal or vertical rotation times. For example, left():

```
// globals
// times for the gun to move a maximum amount in one direction
private final static int LEFT_MAX_TIME = 4900;   // in ms
private final static int RIGHT_MAX_TIME = 8900;

private final static int MIN_PERIOD = 100;  // in ms
      // time for the gun to move the smallest distance

private int horizontalTime = 0;       // left is -ve; right is +ve


public boolean left(int period)
// move left for period ms
{
  if (period < MIN_PERIOD) {
    System.out.println("  left too small: " + period);
    return false;
  }
  else if (horizontalTime-period > -1*LEFT_MAX_TIME) {
    System.out.println("  left: " + period);
    sendCommand(0x04, period);
    horizontalTime -= period;
    return true;
  }
  else {    // reached the left rotation limit
    System.out.println("  left limit");
    return false;
  }
}  // end of left()
```

left(), right(), up(), and down() all have the same three-way branch structure. The first branch rejects a rotation with too small a time period. The second branch carries out the turn and updates the relevant global ongoing rotation time. In left(), if the time required exceeds the time limit for turning left (LEFT_MAX_TIME), then the operation is rejected by the third branch. The right() method is similar but uses RIGHT_MAX_TIME for its time testing.

up() and down() are much the same, but manipulate the ongoing vertical rotation time, as shown in up():

```
// globals
```



```
// times for the gun to move a maximum amount in one direction
private final static int UP_MAX_TIME = 1900;
private final static int DOWN_MAX_TIME = 900;

private final static int MIN_PERIOD = 100;  // in ms
      // time for the gun to move the smallest distance

private int verticalTime = 0;         // down is -ve; up is +ve


public boolean up(int period)
// move up for period ms
{
  if (period < MIN_PERIOD) {
    System.out.println("  up too small: " + period);
    return false;
  }
  else if (verticalTime+period < UP_MAX_TIME) {
    System.out.println("  up: " + period);
    sendCommand(0x02, period);
    verticalTime += period;
    return true;
  }
  else {    // reached the up rotation limit
    System.out.println("  up limit");
    return false;
  }
}  // end of up()
```

7.2. Resetting

The MissileLauncher.reset() method uses the two time globals, horizontalTime and verticalTime to undo the rotations, returning the launcher to its start position.

```
// globals
private int horizontalTime = 0;       // left is -ve; right is +ve
private int verticalTime = 0;         // down is -ve; up is +ve


public void reset()
{
  if (horizontalTime < 0)   // -ve so on left
    right(-1*horizontalTime);
  else   // +ve so on right
    left(horizontalTime);

  if (verticalTime < 0)   // -ve so facing down
    up(-1*verticalTime);
  else   // +ve so facing up
    down(verticalTime);
}  // end of reset()
```

7.3. From Pixel to Time-based Rotations

The MotionLauncher application deals in pixel distances, and so will not directly call left(), right(), up(), and down() because they deal in time periods. Instead

MotionLauncher invokes MissileLauncher.moveHorizontal() with a x-axis pixel length and MissileLauncher.moveVertical () with a y-axis pixel length. The (x, y) coordinates use the normal Java axes, so positive x is to the right and positive y is down the image.

```
// globals
// time for the gun to move across the screen
private final static int WIDTH_MAX_TIME = 2900;   // in ms
private final static int HEIGHT_MAX_TIME = 2900;

private int scrWidth, scrHeight;
    /* dimensions of the screen (I assume they're the same
       width/height ratio as SCR_WIDTH/SCR_HEIGHT */


public boolean moveHorizontal(int x)
// convert x-axis dist into time-period call to left() or right()
{
  int period;
  if (x < 0) {  // x goes to the left in the webcam image
    period = (int) Math.round(((double) -x / scrWidth) *
                                          WIDTH_MAX_TIME);
    return left(period);   // has moved left?
  }
  else {   // x is to right
    period = (int) Math.round(((double) x / scrWidth) *
                                         WIDTH_MAX_TIME);
    return right(period);   // has moved right?
  }
}  // end of moveHorizontal()


public boolean moveVertical(int y)
// convert y-axis dist into time-period call to up() or down()
{
  int period;
  if (y > 0) {  // y is down the webcam image
    period = (int) Math.round(((double) y / scrHeight) *
                                         HEIGHT_MAX_TIME);
    return down(period);   // has moved down?
  }
  else {   // y is up
    period = (int) Math.round(((double) -y / scrHeight) *
                                          HEIGHT_MAX_TIME);
    return up(period);  // has moved up?
  }
}  // end of moveVertical()
```

The calculation of the period uses travel times that I obtained rotating across a 640 x 480 size webcam image. These will be the same for any image with the same width/height ratio.

8. Creating the MotionLauncher application

The MotionLauncher application shown back in Figure 1 is a combination of the MotionDetector application of the previous chapter and the MissileLauncher class just described. This can be seem from the application's class diagrams in Figure 27.

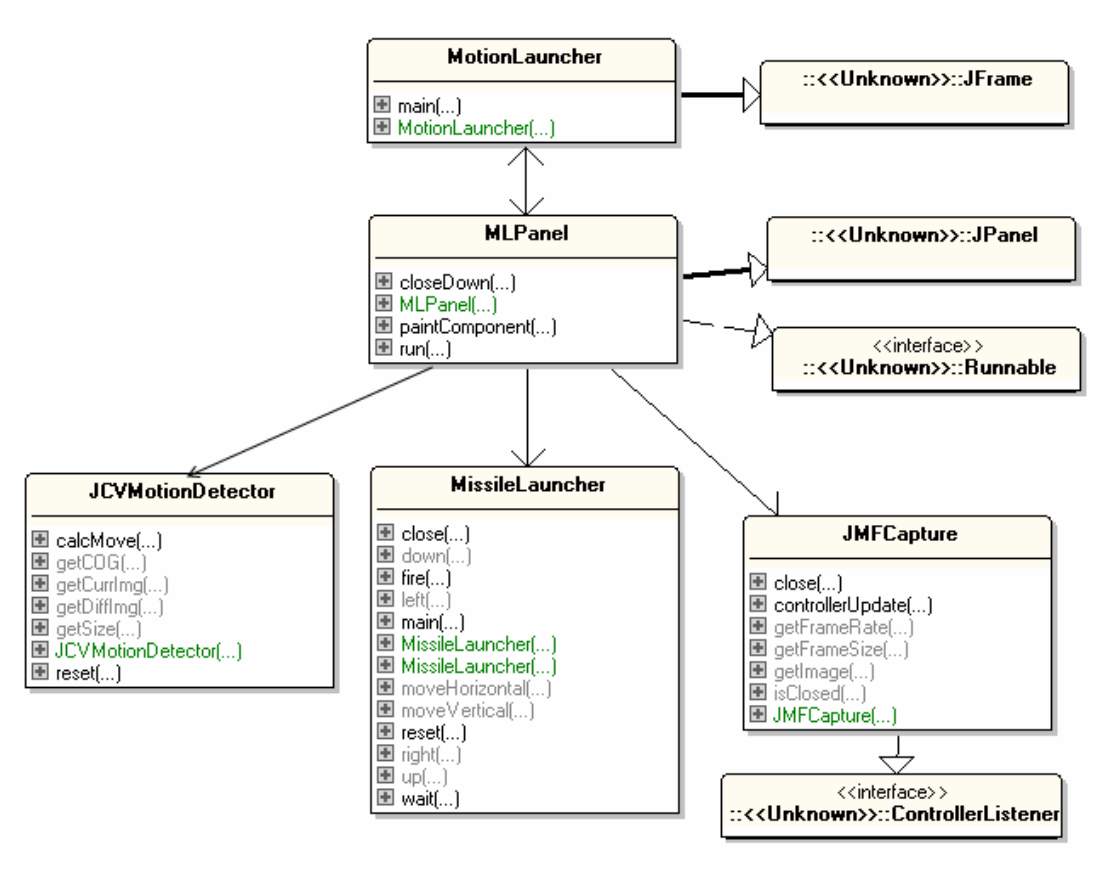

Figure 27. Class Diagrams for the MotionLauncher Application.

The top-level JFrame and JPanel classes, MotionLauncher and MLPanel, are versions of the MotionDetector and MotionPanel classes of the previous chapter. The JMFCapture class is unchanged, and JCVMotionDetector has one small extra method, reset().

In Figure 28, I've superimposed the classes of Figure 27 over the third-party libraries from Figure 4, to show how the application's functionality is divided between the classes.

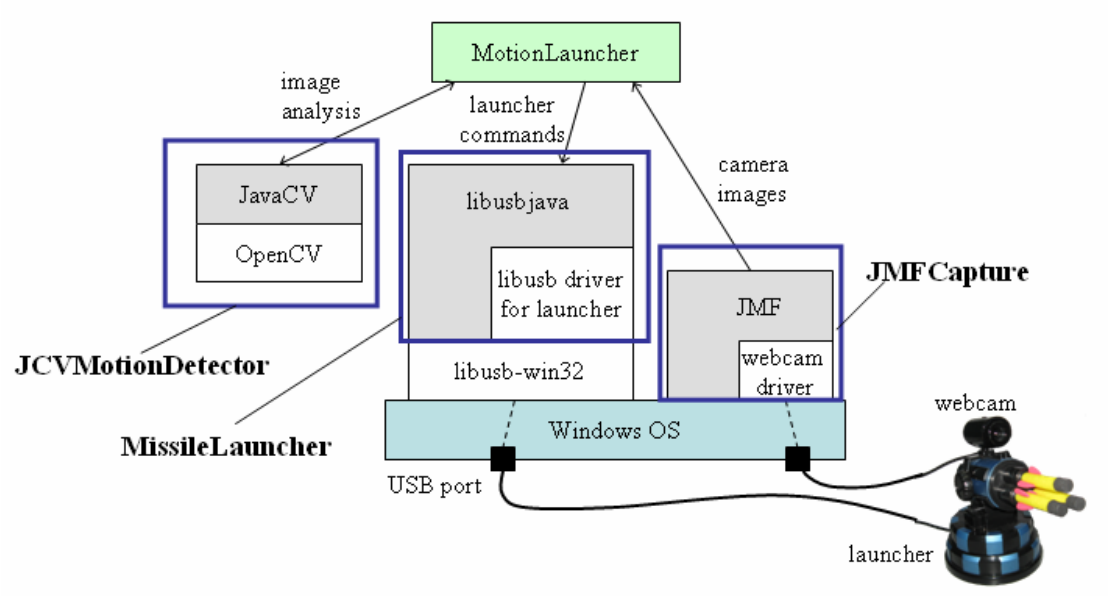

Figure 28. The MotionLauncher Classes and their Libraries.

8.1. Rendering the Launcher's View

The MLPanel class loops inside its run() method, grabbing an image, finding its center-of-gravity (COG), and uses the COG to move the launcher. Much of the coding (imaging, analysis) is the same as in the last chapter's MotionPanel class.

The new element is the positioning of the launcher. Initially, it points toward the center of the webcam image (this isn't quite true, but near enough). When a COG is calculated, the launcher should be moved to point at it. The next webcam image will show a new view of the scene, since the camera is mounted on top of the launcher, and the new image's center will coincide with the COG position.

There's a tricky dependency hidden in the above description. JCVMotionDetector finds a COG by comparing the current webcam image with the previous one, the assumption being that any differences are due to movement in the scene. However, when the launcher moves to a new position, the webcam image will be very different from the previous one. A COG calculation using the previous, now out-of-date, image will give incorrect results. This means that after the launcher has moved, its necessary to generate a new previous image.

Another problem is speed. In previous examples, the panel has been updated at least 10 times/second, controlled by a global DELAY constant set to 100 ms. But each iteration must now include time to move the launcher, and generate new webcam images. Launcher movement is particularly slow, often taking 1-2 seconds.

One way of handling slow processing is to move it into a separate thread where it can be as lethargic as it likes without affecting the rest of the application (an approach I'll use for face recognition in NUI Chapter 8). The drawback is the complexity of writing threads which share data (in this case, webcam images). Instead I'll go for a simpler fix – reducing the frequency of COG analysis and launcher movement. These tasks

will still take 1-2 seconds, but occur less often than once every rendering iteration. From the user's point of view, panel rendering will be fast most of the time, with occasional slowdowns.

The run() method in MLPanel shows how this design is implemented:

```
// in MLPanel
// globals
private static final int DELAY = 500;     // was 100
         // time (ms) between redraws of the panel

private static final int MOVE_DELAY = 1000;
        // time (ms) between moves


private JFrame top;
private BufferedImage image = null;  // current webcam snap
private JMFCapture camera;
private volatile boolean isRunning;

// used for the average ms snap time information
private int imageCount = 0;
private long totalTime = 0;


public void run()
{
  camera = new JMFCapture();

  BufferedImage im = camera.getImage();
  JCVMotionDetector md = new JCVMotionDetector(im);
       // create motion detector

  MissileLauncher launcher = null;

  // update panel and window sizes to fit video's frame size
  Dimension frameSize = camera.getFrameSize();
  if (frameSize != null) {
    setPreferredSize(frameSize);
    top.pack();   // resize and center JFrame
    top.setLocationRelativeTo(null);

    launcher = new MissileLauncher(frameSize.width,
                                   frameSize.height);

    centerPoint = new Point(frameSize.width/2, frameSize.height/2);
  }

  long duration;
  isRunning = true;
  long moveTime = System.currentTimeMillis();
  while (isRunning) {
    long startTime = System.currentTimeMillis();

    im = camera.getImage();  // take a snap
    if (im == null) {
    System.out.println("Problem loading image " + (imageCount+1));
      duration = System.currentTimeMillis() - startTime;
    }
    else {
```



```
      image = im;   // only update image if it contains something
      imageCount++;

      md.calcMove(im);    // update detector with new image

      if ((System.currentTimeMillis() - moveTime) >= MOVE_DELAY) {
        reactToCOG(md, launcher);  // use COG to move launcher
        moveTime = System.currentTimeMillis();
      }

      duration = System.currentTimeMillis() - startTime;
      totalTime += duration;
      repaint();
    }

    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);  // wait until DELAY time
      }
      catch (Exception ex) {}
    }
  }

  launcher.close();
  camera.close();    // close down the camera
}  // end of run()
```

The JCVMotionDetector object is created and used in the same way as in the previous chapter. Most of the new code is in reactToCOG(), executed by the small if-test after the call to calcMove(). Also, reactToCOG() is only called if enough time has passed since the launcher was last moved.

The DELAY constant has also been changed; formerly it was set at 100 ms, but it is now 500 ms, which makes the delays caused by launcher movement less noticeable.

A MissileLauncher object is created inside run(), and closed down when the display loop finishes. MissileLauncher is unmodified from its description in the last section.

8.2. Reacting to Movement

reactToCOG() converts a COG position into launcher movement, a reset of the JCVMotionDetector object, and a possible missile firing.

```
// globals
private static final int FIRE_MOVES = 6;
              // no of consecutive moves before firing


private int numConseqMoves = 0;     // number of consecutive moves


private void reactToCOG(JCVMotionDetector md,
                                MissileLauncher launcher)
{ Point cogPt = md.getCOG();   // get new COG
  if (cogPt == null) {
    System.out.println("No new COG point found");
    numConseqMoves = 0;
    return;
```



```
  }
  else
    System.out.println("COG: (" + cogPt.x + ", " + cogPt.y + ")");

  // calculate offset of COG from center
  int xOffset = cogPt.x - centerPoint.x;
  int yOffset = cogPt.y - centerPoint.y;

  // move launcher so COG is at center spot
  boolean hasMovedHoriz = launcher.moveHorizontal(xOffset);
  boolean hasMovedVert = launcher.moveVertical(yOffset);

  if (hasMovedHoriz || hasMovedVert) {
    image = camera.getImage();     // update panel's image
    md.reset(image);               // reset JCVMotionDetector
    numConseqMoves++;
    if (numConseqMoves == FIRE_MOVES) {
      launcher.fire();
      numConseqMoves = 0;
    }
  }
  else // no move this time so reset numConseqMoves counter
    numConseqMoves = 0;
}  // end of reactToCOG()
```

reactToCOG() generate a new COG point, and sees how far it is from the center of the panel. It calls MissileLauncher.moveHorizontal() and MissileLauncher. moveVertical() with the calculated pixel offsets. If the launcher is moved, then the panel and the JCVMotionDetector object must be assigned a new view of the scene.

A niggly question is when to fire a missile? As usual, I've gone for a simple answer based on a counter, numConseqMoves, which records the number of consecutive launcher moves. If this counter reaches FIRE_MOVES, then firing is initiated. The counter will be reset to 0 if no COG point was detected in the reactToCOG() call or the calculated COG offset was too small to make the launcher move.

MissileLauncher.fire() is the slowest launcher operation, taking nearly 7 seconds to fire a missile. Even worse, the whole thing may be a complete waste of time if all the missiles have already been released (and the user hasn't reloaded them). MissileLauncher.fire() is another good reason for moving launcher manipulation into a separate thread.

8.3. Resetting the JCVMotionDetector Object

The JCVMotionDetector class is virtually unchanged from the version I described in the previous chapter. The only significant change is a new reset() method:

```
// in JCVMotionDetector
// globals
private static final int MAX_PTS = 2;     // was 5
     // size of cogPoints[] array

private IplImage prevImg, currImg;     // grayscale images

private Point[] cogPoints;   // for smoothing COG point values
private int ptIdx, totalPts;
```



```
public void reset(BufferedImage frame)
{
  ptIdx = 0;
  totalPts = 0;
  prevImg = convertFrame(frame);
  currImg = null;
}  // end of reset()
```

The reset affects the COG points array and the previous and current images. The points array is treated as being empty by setting its ptIdx and totalPts counters to 0. This reflects the idea that the 'old' COG points were for the launcher's old position, and so should be discarded. The previous image is set to the new view, and the current image will be recalculated when JCVMotionDetector.calcMove() is next called.

I also reduced the size of the COG points array in JCVMotionDetector to two elements. If you recall, the purpose of this array is to store old COG values so smoothing could combine the current and previous points. This removes some of the jitter from the COG position at the expense of making the COG slower to change position when there's rapid movement.

After experimenting with MotionLauncher, it turns out that COG responsiveness is more desirable than smoothness, so I reduced the size of the points array.

8.4. Some Issues with the Camera

One of the long standing restrictions of JMF is that is only supports one webcam per JVM (according to its documentation). In practice, it seems that even if multiple JVMs are running, JMF will still only see a single camera, even when MS Windows can quite happily switch between them.

What has this got to do with MotionLauncher, which only uses the camera stuck atop the launcher? The problem is that most modern PCs come with a built-in webcam, and so plugging the launcher's camera into the machine's USB port will bring the number to two, creating problems for JMF.

This lack of JMF functionality, combined with the likelihood that the issue will never be fixed, is what encourages people to turn to more modern, functionally more advanced, supported, third-party libraries. An increasingly popular one is FMJ (http://fmj-sf.net/) which bills itself as an open-source alternative to JMF, that aims to be API-compatible. According to its help forum, FMJ does support multiple cameras.

I didn't go the FMJ route since there's a fairly simple fix in Windows for my problem – I can temporarily disable the PC's webcam driver via Window's Device Manager, leaving only the Dream Cheeky camera. It's also necessary to register the camera with the JMF Registry application.

9. More Information on the Dream Cheeky Launcher

Investigating the USB protocol for the Dream Cheeky Launcher involved USBDeview (free from http://www.nirsoft.net/utils/usb_devices_view.html) and USBTrace (free 15 day trial from http://www.sysnucleus.com/), and the development of code on top of LibusbJava (http://libusbjava.sourceforge.net/) and libusb-win32 (http://sourceforge.net/apps/trac/libusb-win32/wiki). In some respects, the investigation process and tools are more important than what I found out about the launcher, since I can apply these techniques to other USB gadgets. I'll be doing just that when I start playing with a toy robot arm in NUI Chapter 6.

I was a bit miffed not to be able to read interrupt transfers (anyone who gets this working in Java, please contact me). However, it seems that similar problems have affected people programming the launcher in other languages and platforms.

Due to its high fun quota, there's a lot of advice out of the Web about how to interface to the launcher. Perhaps the best first stop is David Wilson's blog (http://dgwilson.wordpress.com/). Look for the blog section headings entitled "USB Missile" and "Download" that hold lots of C code for several versions of the launcher on the Mac and Windows; the Windows information can be accessed directly at http://dgwilson.wordpress.com/windows-missile-launcher/. One nice thing about his blog posts, are the informative comments by readers.

Wilson makes an important point: that different launcher versions (at least four from Dream Cheeky, not counting other manufacturers) utilize similar USB commands, but *not* the same ones. For example, my analysis of Dream Cheeky's MSN Missile Launcher is not quite the same as their current Storm O.I.C launcher (http://www.dreamcheeky.com/storm-oic-missile-launcher)

Wilson doesn't consider Linux. For that platform, you should visit Matthias Vallentin's blog post "A Linux kernel driver for the Dream Cheeky USB missile launcher" (http://matthias.vallentin.net/2007/04/writing-linux-kernel-driver-for-unknown.html). It includes a detailed explanation of the reverse engineering of the protocol, and good comments from readers. The driver, called ML-Driver, is available from https://github.com/mavam/ml-driver.

Another Linux version, written in Perl, is described in an article in *Linux Magazine* called "Replacing Tin Soldiers" by Michael Schilli, available at http://www.linux-magazine.com/w3/issue/103/Perl.pdf

For Python programmers, try "Python Interfacing a USB Missile Launcher" by Pedram Amini at http://dvlabs.tippingpoint.com/blog/2009/02/12/python-interfacing-a-usb-missile-launcher. The code is at http://dvlabs.tippingpoint.com/pub/pamini/ped_missile.py. Amini traces the USB protocol, and also disassembles Dream Cheeky's support library, USBHID.dll.

Chris Smith utilizes F# in "Being an Evil Genius with F# and .NET" at http://blogs.msdn.com/b/chrsmith/archive/2010/01/24/being-an-evil-genius-with-f-and-net.aspx. He adds speech control using .NET's System.Speech API, something I'll be doing in Java in NUI Chapter 11.

Fancy a Wii? Then take a look at "Wiimotely Controlled Wee Little Rockets" by Chad Z. Hower at http://www.kudzuworld.com/blogs/Tech/WiiRockets.en.aspx and http://rocket.codeplex.com/. He uses C# and two external libraries, WiimoteLib and USB HID, for keyboard and Wiimote (Wii remote) support.

The launcher makes a good university computer engineering project. For example, see "Run, Stephen, Run: Shoot First, Ask Questions Later" by Andrew Bui et al. at Columbia (http://www.cs.columbia.edu/~sedwards/classes/2010/4840/reports/RSR.pdf). They develop controller hardware and software for a Altera DE2 Board, and look at ballistics and triangulation issues. There's also the Group05 project from UNSW (http://cgi.cse.unsw.edu.au/~cs4411/wiki/index.php?title=Group05_Project) that combines the launcher with robots that search, aim and fire at each other autonomously. They employ Haar classifiers in OpenCV and ARToolkit, topics I'll be considering in NUI Chapters 8 and 19. The report includes lots of fun pictures.

What came as a bit of a surprise were the many open source libraries for toy launchers, written in different languages (but not in Java) I found five libraries at http://sourceforge.net/ and another five at http://code.google.com/. You should search with keywords such as "missile" and "launcher".

