



Author: Jon Barnett  
Date: 3<sup>rd</sup> April 2004

## Targus PAKP003 Mouse: Making it work in Linux

I recently had to get a Targus wireless mini optical mouse working with a Slackware 9.1 distribution. The Linux kernel version was 2.4.24. The Targus mouse was part of the Targus PAKP003 package that consisted of the USB wireless receiver, the mouse and a wireless keypad.

The first thing I noticed was that the wireless receiver when plugged in, was not recognised and no driver was registered for it. The following shows the boot messages from a 'dmesg' capture.

```
usb.c: registered new driver usbdevfs
usb.c: registered new driver hub
uhci.c: USB Universal Host Controller Interface driver v1.1
PCI: Setting latency timer of device 00:1f.2 to 64
uhci.c: USB UHCI at I/O 0xdce0, IRQ 10
usb.c: new USB bus registered, assigned bus number 1
hub.c: USB hub found
hub.c: 2 ports detected
hub.c: new USB device 00:1f.2-1, assigned address 2
usb.c: USB device 2 (vend/prod 0xa91/0x3801) is not claimed by any
active driver
```

This is actually fine as the receiver only ferries signal from the wireless devices and translates them to something the standard USB device drivers can understand – normally, we would expect the signals to meet the human interface device (HID) specifications for the universal serial bus (USB).

My Dell Inspiron 8100 has the Intel USB host controller known as the Universal Host Controller Interface (UHCI). Your system may have the Open Host Controller Interface (OHCI). Use the module that works for your setup.

You can check the USB devices detected by the 'lsusb' command.

```
Bus 001 Device 001: ID 0000:0000 Virtual Hub
Bus 001 Device 002: ID 0a91:3801
```

The vendor ID is 0a91 and the device ID is 3801. According to the Linux USB website, the vendor ID belongs to Globlink Technology, Inc. You can find out more information on this and Linux USB in general at <http://www.linux-usb.org>.

In order for normal programs in Linux to use the signals coming from these USB-connected devices, you need device drivers to translate these inputs from the Linux HID layer. The modern Linux distributions should be able to determine the correct drivers through the hotplug utility. Otherwise, you would achieve this through /etc/modules.conf.

The relevant output from 'dmesg' would be something similar to the following:

```
usb.c: registered new driver usbkbd
input0: wireless inc tw wireless usb device on usb1:2.0
usbkbd.c: :USB HID Boot Protocol keyboard driver
usb.c: registered new driver hiddev
usb.c: registered new driver hid
input,hiddev0: USB HID v1.10 Mouse [wireless inc tw wireless usb
device] on usb1:2.1
hid-core.c: v1.8.1 Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>
hid-core.c: USB HID support drivers
mice: PS/2 mouse device common for all mice
```

The important things to note here are that a HID keyboard device driver and a HID mouse device driver was loaded along with the Linux HID core. The raw output from the mouse will be sent to /dev/usb/hiddev0. The particular hiddev character interface will vary, depending on whether you have any other HID input devices for your system. Should the device node not exist for your system, you will need to create it. There are instructions with the Linux 2.4.24 kernel source, in the Documentation/usb/hiddev.txt.

```
mknod /dev/usb/hiddev0 c 180 96
```

The mouse character stream will bind to one of the standard mouse character device streams. In my case, this was /dev/input/mouse0. Additionally as shown in the 'dmesg' output, the HID mouse stream was directed through to the USB PS/2 character stream at /dev/input/mice. All USB mice are available through this device node. This makes it easy for programs such as Xfree86 to use any PS/2 compatible USB mouse, without having to define a separate InputDevice for every USB mouse.

Check your Linux module stack. An 'lsmod' should give you results similar to the following:

Module	Size	Used by	Tainted: P
keybdev	2112	0 (unused)	
mousedev	4532	1	
hid	21924	0 (unused)	
usbkbd	3640	0 (unused)	
input	3328	0 [keybdev mousedev hid usbkbd]	
uhci	25968	0 (unused)	
usbcore	62976	1 [hid usbkbd uhci]	

Having set this up, I was ready to test the mouse and the keypad. The keyboard worked fine after I initialized the receiver to the keypad and the mouse. The keypad was immediately able to generate output at the command line without any additional work.

## **Problems with the mouse**

I tested the mouse first by viewing the response from the raw HID stream. Performing the following:

```
cat /dev/usb/hiddev0
```

and moving the mouse around, clicking the buttons and rotating the mouse wheel showed output for each event.

I configured gpm, the cut and paste mouse utility to use the Targus mouse. In Slackware, you do this by editing `/etc/rc.d/rc.gpm` and initializing gpm so that it uses the USB PS/2 output stream at `/dev/input/mice` and interprets the input as an Intellimouse PS/2 compatible protocol.

```
/usr/sbin/gpm -m /dev/input/mice -t imps2
```

On running gpm, the utility ran without a problem at initialization. Should you choose the wrong device, or the wrong protocol, the program will usually fail to start properly. I then discovered that while I could move the mouse cursor around, I could not select anything.

Now since I was unfamiliar about the USB device driver architecture in Linux so I had to research the processing of the USB mouse input. Everything for USB input devices comes through `drivers/input/input.c` from the Linux 2.4.24 source. I modified this first to determine what was happening. The button pushes are key events so I put some test code around that.

```
void input_event(struct input_dev *dev, unsigned int type,
    unsigned int code, int value)
{
    struct input_handle *handle = dev->handle;

    /*
     * Filter non-events, and bad input values out.
     */

    if (type > EV_MAX || !test_bit(type, dev->evbit))
        return;

    switch (type) {

        case EV_KEY:

            printk(KERN_INFO "IN: %d %d\n", code, value); //added
            if (code > KEY_MAX || !test_bit(code, dev->keybit) ||
                !test_bit(code, dev->key) == value)
                return;
```

KEY\_MAX was defined as 511. I installed the modified and recompiled modules ('make modules;make modules\_install' is the easiest way of doing this), rebooted and clicked the mouse buttons. I ran 'dmesg' to see the messages I generated to the kernel ring buffer.

The code for any of the mouse button events was 512. Clearly, this was the source of the problem as it meant that these events were being filtered out by the input module. The value was 1 when the button was clicked and 0 when the button was released.

I looked for code that called input\_event. Since the Linux USB documentation indicated that this was the HID module, I switched to drivers/usb. The calls were made in hid-input.c by the procedure hidinput\_hid\_event. The code here didn't have anything of any real interest. It did some processing but it wasn't the button event trap. I did add some trace code in the configuration procedure, hidinput\_configure\_usage to determine the codes to which the buttons were bound.

```
case HID_UP_BUTTON:

    usage->code = ((usage->hid - 1) & 0xf) + 0x100;
    usage->type = EV_KEY; bit = input->keybit; max = KEY_MAX;

    switch (field->application) {
        case HID_GD_GAMEPAD:    usage->code += 0x10;
        case HID_GD_JOYSTICK:   usage->code += 0x10;
        case HID_GD_MOUSE:      usage->code += 0x10; break;
        default:
            if (field->physical == HID_GD_POINTER)
                usage->code += 0x10;
            break;
        if ( field->application==HID_GD_MOUSE) //added
            printk(KERN_INFO "CODE: %d\n", usage->code); //added
    }
    break;
```

I installed the recompiled modules, rebooted and clicked the mouse buttons. I ran 'dmesg' to see the configuration mapping generated through the kernel messages. Button 1 corresponded to code 272, button 2 to code 273 and button 3 to code 274. I now knew which codes were supposed to be associated with the mouse buttons.

Turning back to the problem of tracking the input mappings for the events, I looked at hid-core.c as this called the hidinput\_hid\_event procedure. I also looked at drivers/usb/hid.h to determine the construction of the structure hid\_usage. From this information, I generated some value traces to determine empirically the mouse button events.

```

static void hid_process_event(struct hid_device *hid,
    struct hid_field *field, struct hid_usage *usage, __s32 value)
{
    hid_dump_input(usage, value);
    if (usage->type==EV_KEY) //added
        printk(KERN_INFO "EV: %d %d %d\n", usage->hid, usage->code,
            value); //added
    if (hid->claimed & HID_CLAIMED_INPUT)
        hidinput_hid_event(hid, field, usage, value);
    if (hid->claimed & HID_CLAIMED_HIDDEV)
        hiddev_hid_event(hid, field, usage, value);
}

```

From these results, and the corresponding trace events in drivers/input/input.c I was able to construct the following:

Button	usage->hid	usage->code	valid usage->code
1	589825 (9001H)	512	272 (110H)
2	589826 (9002H)	512	273 (111H)
3	589827 (9003H)	512	274 (112H)

Since the rest of the HID operation required much more study and was not necessarily of use to me as I only wanted to get the mouse operational, I decided to make the modification for the mapping in this area.

```

static void hid_process_event(struct hid_device *hid,
    struct hid_field *field, struct hid_usage *usage, __s32 value)
{
    hid_dump_input(usage, value);
    if (usage->hid==589825)
        usage->code = 272;
    if (usage->hid==589826)
        usage->code = 273;
    if (usage->hid==589827)
        usage->code = 274;
    if (hid->claimed & HID_CLAIMED_INPUT)
        hidinput_hid_event(hid, field, usage, value);
    if (hid->claimed & HID_CLAIMED_HIDDEV)
        hiddev_hid_event(hid, field, usage, value);
}

```

With these changes made I recompiled the modules, installed them and rebooted the Linux system. I tested operation of the mouse in Dropline GNOME. I modified XF86Config-4 with the following definitions:

```
Section "InputDevice"
    Identifier "Mouse1"
    Driver      "mouse"
    Option      "Protocol"      "IMPS/2"
    Option      "Device"        "/dev/input/mice"
    Option      "CorePointer"
    Option      "ZAxisMapping"  "4 5"
    Option      "Buttons"       "5"
EndSection
```

```
Section "ServerLayout"
    Identifier "AGP"
    Screen     "Screen AGP"
    InputDevice "Mouse1"
    InputDevice "Keyboard1" "CoreKeyboard"
EndSection
```

Both the mouse and the keypad operated normally, and the GNOME system responded to the button clicks of the mouse as expected. While the modification was perhaps less than elegant, this was satisfactory for my needs as I did not intend to learn more about USB and HID than I absolutely needed to get my mouse working.

As a final cleanup for production, I removed the printk kernel messages, rebuilt the modules and installed the modules.