The Linux input driver subsystem

Release

The kernel development community

Nov 02, 2017

CONTENTS

Contents:

CHAPTER

ONE

LINUX INPUT SUBSYSTEM USERSPACE API

Table of Contents

Introduction

Architecture

Input subsystem a collection of drivers that is designed to support all input devices under Linux. Most of the drivers reside in drivers/input, although quite a few live in drivers/hid and drivers/platform.

The core of the input subsystem is the input module, which must be loaded before any other of the input modules - it serves as a way of communication between two groups of modules:

Device drivers

These modules talk to the hardware (for example via USB), and provide events (keystrokes, mouse movements) to the input module.

Event handlers

These modules get events from input core and pass them where needed via various interfaces - keystrokes to the kernel, mouse movements via a simulated PS/2 interface to GPM and X, and so on.

Simple Usage

For the most usual configuration, with one USB mouse and one USB keyboard, you'll have to load the following modules (or have them built in to the kernel):

```
input
mousedev
usbcore
uhci_hcd or ohci_hcd or ehci_hcd
usbhid
hid_generic
```

After this, the USB keyboard will work straight away, and the USB mouse will be available as a character device on major 13, minor 63:

```
crw-r--r--   1 root     root      13,  63 Mar 28 22:45 mice
```

This device usually created automatically by the system. The commands to create it by hand are:

```
cd /dev
mkdir input
mknod input/mice c 13 63
```

After that you have to point GPM (the textmode mouse cut&paste tool) and XFree to this device to use it - GPM should be called like:

```
gpm -t ps2 -m /dev/input/mice
```

And in X:

```
Section "Pointer"
    Protocol    "ImPS/2"
    Device      "/dev/input/mice"
    ZAxisMapping 4 5
EndSection
```

When you do all of the above, you can use your USB mouse and keyboard.

Detailed Description

Event handlers

Event handlers distribute the events from the devices to userspace and in-kernel consumers, as needed.

evdev

evdev is the generic input event interface. It passes the events generated in the kernel straight to the program, with timestamps. The event codes are the same on all architectures and are hardware independent.

This is the preferred interface for userspace to consume user input, and all clients are encouraged to use it.

See *Event interface* for notes on API.

The devices are in /dev/input:

```
crw-r--r--   1 root     root      13,  64 Apr  1 10:49 event0
crw-r--r--   1 root     root      13,  65 Apr  1 10:50 event1
crw-r--r--   1 root     root      13,  66 Apr  1 10:50 event2
crw-r--r--   1 root     root      13,  67 Apr  1 10:50 event3
...
```

There are two ranges of minors: 64 through 95 is the static legacy range. If there are more than 32 input devices in a system, additional evdev nodes are created with minors starting with 256.

keyboard

keyboard is in-kernel input handler and is a part of VT code. It consumes keyboard keystrokes and handles user input for VT consoles.

mousedev

mousedev is a hack to make legacy programs that use mouse input work. It takes events from either mice or digitizers/tablets and makes a PS/2-style (a la /dev/psaux) mouse device available to the userland.

Mousedev devices in /dev/input (as shown above) are:

```
crw-r--r--   1 root     root      13,  32 Mar 28 22:45 mouse0
crw-r--r--   1 root     root      13,  33 Mar 29 00:41 mouse1
crw-r--r--   1 root     root      13,  34 Mar 29 00:41 mouse2
crw-r--r--   1 root     root      13,  35 Apr  1 10:50 mouse3
...
...
crw-r--r--   1 root     root      13,  62 Apr  1 10:50 mouse30
crw-r--r--   1 root     root      13,  63 Apr  1 10:50 mice
```

Each mouse device is assigned to a single mouse or digitizer, except the last one - mice. This single character device is shared by all mice and digitizers, and even if none are connected, the device is present. This is useful for hotplugging USB mice, so that older programs that do not handle hotplug can open the device even when no mice are present.

CONFIG_INPUT_MOUSEDEV_SCREEN_[XY] in the kernel configuration are the size of your screen (in pixels) in XFree86. This is needed if you want to use your digitizer in X, because its movement is sent to X via a virtual PS/2 mouse and thus needs to be scaled accordingly. These values won't be used if you use a mouse only.

Mousedev will generate either PS/2, ImPS/2 (Microsoft IntelliMouse) or ExplorerPS/2 (IntelliMouse Explorer) protocols, depending on what the program reading the data wishes. You can set GPM and X to any of these. You'll need ImPS/2 if you want to make use of a wheel on a USB mouse and ExplorerPS/2 if you want to use extra (up to 5) buttons.

joydev

joydev implements v0.x and v1.x Linux joystick API. See *Programming Interface* for details.

As soon as any joystick is connected, it can be accessed in /dev/input on:

```
crw-r--r--   1 root     root      13,   0 Apr  1 10:50 js0
crw-r--r--   1 root     root      13,   1 Apr  1 10:50 js1
crw-r--r--   1 root     root      13,   2 Apr  1 10:50 js2
crw-r--r--   1 root     root      13,   3 Apr  1 10:50 js3
...
```

And so on up to js31 in legacy range, and additional nodes with minors above 256 if there are more joystick devices.

Device drivers

Device drivers are the modules that generate events.

hid-generic

hid-generic is one of the largest and most complex driver of the whole suite. It handles all HID devices, and because there is a very wide variety of them, and because the USB HID specification isn't simple, it needs to be this big.

Currently, it handles USB mice, joysticks, gamepads, steering wheels keyboards, trackballs and digitizers.

However, USB uses HID also for monitor controls, speaker controls, UPSs, LCDs and many other purposes.

The monitor and speaker controls should be easy to add to the hid/input interface, but for the UPSs and LCDs it doesn't make much sense. For this, the hiddev interface was designed. See Documentation/hid/hiddev.txt for more information about it.

The usage of the usbhid module is very simple, it takes no parameters, detects everything automatically and when a HID device is inserted, it detects it appropriately.

However, because the devices vary wildly, you might happen to have a device that doesn't work well. In that case #define DEBUG at the beginning of hid-core.c and send me the syslog traces.

usbmouse

For embedded systems, for mice with broken HID descriptors and just any other use when the big usbhid wouldn't be a good choice, there is the usbmouse driver. It handles USB mice only. It uses a simpler HIDBP protocol. This also means the mice must support this simpler protocol. Not all do. If you don't have any strong reason to use this module, use usbhid instead.

usbkbd

Much like usbmouse, this module talks to keyboards with a simplified HIDBP protocol. It's smaller, but doesn't support any extra special keys. Use usbhid instead if there isn't any special reason to use this.

psmouse

This is driver for all flavors of pointing devices using PS/2 protocol, including Synaptics and ALPS touchpads, Intellimouse Explorer devices, Logitech PS/2 mice and so on.

atkbd

This is driver for PS/2 (AT) keyboards.

iforce

A driver for I-Force joysticks and wheels, both over USB and RS232. It includes Force Feedback support now, even though Immersion Corp. considers the protocol a trade secret and won't disclose a word about it.

Verifying if it works

Typing a couple keys on the keyboard should be enough to check that a keyboard works and is correctly connected to the kernel keyboard driver.

Doing a `cat /dev/input/mouse0` (c, 13, 32) will verify that a mouse is also emulated; characters should appear if you move it.

You can test the joystick emulation with the `jstest` utility, available in the joystick package (see *Introduction*).

You can test the event devices with the `evtest` utility.

Event interface

You can use blocking and nonblocking reads, and also select() on the /dev/input/eventX devices, and you'll always get a whole number of input events on a read. Their layout is:

```
struct input_event {
        struct timeval time;
        unsigned short type;
        unsigned short code;
        unsigned int value;
};
```

`time` is the timestamp, it returns the time at which the event happened. Type is for example EV_REL for relative moment, EV_KEY for a keypress or release. More types are defined in include/uapi/linux/input-event-codes.h.

`code` is event code, for example REL_X or KEY_BACKSPACE, again a complete list is in include/uapi/linux/input-event-codes.h.

`value` is the value the event carries. Either a relative change for EV_REL, absolute new value for EV_ABS (joysticks ...), or 0 for EV_KEY for release, 1 for keypress and 2 for autorepeat.

See *Input event codes* for more information about various even codes.

Input event codes

The input protocol uses a map of types and codes to express input device values to userspace. This document describes the types and codes and how and when they may be used.

A single hardware event generates multiple input events. Each input event contains the new value of a single data item. A special event type, EV_SYN, is used to separate input events into packets of input data changes occurring at the same moment in time. In the following, the term “event” refers to a single input event encompassing a type, code, and value.

The input protocol is a stateful protocol. Events are emitted only when values of event codes have changed. However, the state is maintained within the Linux input subsystem; drivers do not need to maintain the state and may attempt to emit unchanged values without harm. Userspace may obtain the current state of event code values using the EVIOCG* ioctls defined in linux/input.h. The event reports supported by a device are also provided by sysfs in class/input/event*/device/capabilities/, and the properties of a device are provided in class/input/event*/device/properties.

Event types

Event types are groupings of codes under a logical input construct. Each type has a set of applicable codes to be used in generating events. See the Codes section for details on valid codes for each type.

- EV_SYN:
  - Used as markers to separate events. Events may be separated in time or in space, such as with the multitouch protocol.
- EV_KEY:
  - Used to describe state changes of keyboards, buttons, or other key-like devices.
- EV_REL:
  - Used to describe relative axis value changes, e.g. moving the mouse 5 units to the left.
- EV_ABS:
  - Used to describe absolute axis value changes, e.g. describing the coordinates of a touch on a touchscreen.
- EV_MSC:
  - Used to describe miscellaneous input data that do not fit into other types.
- EV_SW:
  - Used to describe binary state input switches.
- EV_LED:
  - Used to turn LEDs on devices on and off.

- EV_SND:
  - Used to output sound to devices.
- EV_REP:
  - Used for autorepeating devices.
- EV_FF:
  - Used to send force feedback commands to an input device.
- EV_PWR:
  - A special type for power button and switch input.
- EV_FF_STATUS:
  - Used to receive force feedback device status.

Event codes

Event codes define the precise type of event.

EV_SYN

EV_SYN event values are undefined. Their usage is defined only by when they are sent in the evdev event stream.

- SYN_REPORT:
  - Used to synchronize and separate events into packets of input data changes occurring at the same moment in time. For example, motion of a mouse may set the REL_X and REL_Y values for one motion, then emit a SYN_REPORT. The next motion will emit more REL_X and REL_Y values and send another SYN_REPORT.
- SYN_CONFIG:
  - TBD
- SYN_MT_REPORT:
  - Used to synchronize and separate touch events. See the multi-touch-protocol.txt document for more information.
- SYN_DROPPED:
  - Used to indicate buffer overrun in the evdev client's event queue. Client should ignore all events up to and including next SYN_REPORT event and query the device (using EVIOCG* ioctls) to obtain its current state.

EV_KEY

EV_KEY events take the form KEY_<name> or BTN_<name>. For example, KEY_A is used to represent the 'A' key on a keyboard. When a key is depressed, an event with the key's code is emitted with value 1. When the key is released, an event is emitted with value 0. Some hardware send events when a key is repeated. These events have a value of 2. In general, KEY_<name> is used for keyboard keys, and BTN_<name> is used for other types of momentary switch events.

A few EV_KEY codes have special meanings:

- BTN_TOOL_<name>:

  - These codes are used in conjunction with input trackpads, tablets, and touchscreens. These devices may be used with fingers, pens, or other tools. When an event occurs and a tool is used, the corresponding BTN_TOOL_<name> code should be set to a value of 1. When the tool is no longer interacting with the input device, the BTN_TOOL_<name> code should be reset to 0. All trackpads, tablets, and touchscreens should use at least one BTN_TOOL_<name> code when events are generated.

- BTN_TOUCH:

  BTN_TOUCH is used for touch contact. While an input tool is determined to be within meaningful physical contact, the value of this property must be set to 1. Meaningful physical contact may mean any contact, or it may mean contact conditioned by an implementation defined property. For example, a touchpad may set the value to 1 only when the touch pressure rises above a certain value. BTN_TOUCH may be combined with BTN_TOOL_<name> codes. For example, a pen tablet may set BTN_TOOL_PEN to 1 and BTN_TOUCH to 0 while the pen is hovering over but not touching the tablet surface.

Note: For appropriate function of the legacy mousedev emulation driver, BTN_TOUCH must be the first evdev code emitted in a synchronization frame.

Note: Historically a touch device with BTN_TOOL_FINGER and BTN_TOUCH was interpreted as a touchpad by userspace, while a similar device without BTN_TOOL_FINGER was interpreted as a touchscreen. For backwards compatibility with current userspace it is recommended to follow this distinction. In the future, this distinction will be deprecated and the device properties ioctl EVIOCGPROP, defined in linux/input.h, will be used to convey the device type.

- BTN_TOOL_FINGER, BTN_TOOL_DOUBLETAP, BTN_TOOL_TRIPLETAP, BTN_TOOL_QUADTAP:

  - These codes denote one, two, three, and four finger interaction on a trackpad or touchscreen. For example, if the user uses two fingers and moves them on the touchpad in an effort to scroll content on screen, BTN_TOOL_DOUBLETAP should be set to value 1 for the duration of the motion. Note that all BTN_TOOL_<name> codes and the BTN_TOUCH code are orthogonal in purpose. A trackpad event generated by finger touches should generate events for one code from each group. At most only one of these BTN_TOOL_<name> codes should have a value of 1 during any synchronization frame.

Note: Historically some drivers emitted multiple of the finger count codes with a value of 1 in the same synchronization frame. This usage is deprecated.

Note: In multitouch drivers, the input_mt_report_finger_count() function should be used to emit these codes. Please see multi-touch-protocol.txt for details.

EV_REL

EV_REL events describe relative changes in a property. For example, a mouse may move to the left by a certain number of units, but its absolute position in space is unknown. If the absolute position is known, EV_ABS codes should be used instead of EV_REL codes.

A few EV_REL codes have special meanings:

- REL_WHEEL, REL_HWHEEL:

  - These codes are used for vertical and horizontal scroll wheels, respectively.

EV_ABS

EV_ABS events describe absolute changes in a property. For example, a touchpad may emit coordinates for a touch location.

A few EV_ABS codes have special meanings:

- ABS_DISTANCE:

  - Used to describe the distance of a tool from an interaction surface. This event should only be emitted while the tool is hovering, meaning in close proximity of the device and while the value of the BTN_TOUCH code is 0. If the input device may be used freely in three dimensions, consider ABS_Z instead.
  - BTN_TOOL_<name> should be set to 1 when the tool comes into detectable proximity and set to 0 when the tool leaves detectable proximity. BTN_TOOL_<name> signals the type of tool that is currently detected by the hardware and is otherwise independent of ABS_DISTANCE and/or BTN_TOUCH.

- ABS_MT_<name>:
  - Used to describe multitouch input events. Please see multi-touch-protocol.txt for details.

EV_SW

EV_SW events describe stateful binary switches. For example, the SW_LID code is used to denote when a laptop lid is closed.

Upon binding to a device or resuming from suspend, a driver must report the current switch state. This ensures that the device, kernel, and userspace state is in sync.

Upon resume, if the switch state is the same as before suspend, then the input subsystem will filter out the duplicate switch state reports. The driver does not need to keep the state of the switch at any time.

EV_MSC

EV_MSC events are used for input and output events that do not fall under other categories.

A few EV_MSC codes have special meaning:

- MSC_TIMESTAMP:
  - Used to report the number of microseconds since the last reset. This event should be coded as an uint32 value, which is allowed to wrap around with no special consequence. It is assumed that the time difference between two consecutive events is reliable on a reasonable time scale (hours). A reset to zero can happen, in which case the time since the last event is unknown. If the device does not provide this information, the driver must not provide it to user space.

EV_LED

EV_LED events are used for input and output to set and query the state of various LEDs on devices.

EV_REP

EV_REP events are used for specifying autorepeating events.

EV_SND

EV_SND events are used for sending sound commands to simple sound output devices.

EV_FF

EV_FF events are used to initialize a force feedback capable device and to cause such device to feedback.

EV_PWR

EV_PWR events are a special type of event used specifically for power management. Its usage is not well defined. To be addressed later.

Device properties

Normally, userspace sets up an input device based on the data it emits, i.e., the event types. In the case of two devices emitting the same event types, additional information can be provided in the form of device properties.

INPUT_PROP_DIRECT + INPUT_PROP_POINTER

The INPUT_PROP_DIRECT property indicates that device coordinates should be directly mapped to screen coordinates (not taking into account trivial transformations, such as scaling, flipping and rotating). Non-direct input devices require non-trivial transformation, such as absolute to relative transformation for touchpads. Typical direct input devices: touchscreens, drawing tablets; non-direct devices: touchpads, mice.

The INPUT_PROP_POINTER property indicates that the device is not transposed on the screen and thus requires use of an on-screen pointer to trace user's movements. Typical pointer devices: touchpads, tablets, mice; non-pointer device: touchscreen.

If neither INPUT_PROP_DIRECT or INPUT_PROP_POINTER are set, the property is considered undefined and the device type should be deduced in the traditional way, using emitted event types.

INPUT_PROP_BUTTONPAD

For touchpads where the button is placed beneath the surface, such that pressing down on the pad causes a button click, this property should be set. Common in clickpad notebooks and macbooks from 2009 and onwards.

Originally, the buttonpad property was coded into the bcm5974 driver version field under the name integrated button. For backwards compatibility, both methods need to be checked in userspace.

INPUT_PROP_SEMI_MT

Some touchpads, most common between 2008 and 2011, can detect the presence of multiple contacts without resolving the individual positions; only the number of contacts and a rectangular shape is known. For such touchpads, the semi-mt property should be set.

Depending on the device, the rectangle may enclose all touches, like a bounding box, or just some of them, for instance the two most recent touches. The diversity makes the rectangle of limited use, but some gestures can normally be extracted from it.

If INPUT_PROP_SEMI_MT is not set, the device is assumed to be a true MT device.

INPUT_PROP_TOPBUTTONPAD

Some laptops, most notably the Lenovo 40 series provide a trackstick device but do not have physical buttons associated with the trackstick device. Instead, the top area of the touchpad is marked to show visual/haptic areas for left, middle, right buttons intended to be used with the trackstick.

If INPUT_PROP_TOPBUTTONPAD is set, userspace should emulate buttons accordingly. This property does not affect kernel behavior. The kernel does not provide button emulation for such devices but treats them as any other INPUT_PROP_BUTTONPAD device.

INPUT_PROP_ACCELEROMETER

Directional axes on this device (absolute and/or relative x, y, z) represent accelerometer data. Some devices also report gyroscope data, which devices can report through the rotational axes (absolute and/or relative rx, ry, rz).

All other axes retain their meaning. A device must not mix regular directional axes and accelerometer axes on the same event node.

Guidelines

The guidelines below ensure proper single-touch and multi-finger functionality. For multi-touch functionality, see the multi-touch-protocol.txt document for more information.

Mice

REL_{X,Y} must be reported when the mouse moves. BTN_LEFT must be used to report the primary button press. BTN_{MIDDLE,RIGHT,4,5,etc.} should be used to report further buttons of the device. REL_WHEEL and REL_HWHEEL should be used to report scroll wheel events where available.

Touchscreens

ABS_{X,Y} must be reported with the location of the touch. BTN_TOUCH must be used to report when a touch is active on the screen. BTN_{MOUSE,LEFT,MIDDLE,RIGHT} must not be reported as the result of touch contact. BTN_TOOL_<name> events should be reported where possible.

For new hardware, INPUT_PROP_DIRECT should be set.

Trackpads

Legacy trackpads that only provide relative position information must report events like mice described above.

Trackpads that provide absolute touch position must report ABS_{X,Y} for the location of the touch. BTN_TOUCH should be used to report when a touch is active on the trackpad. Where multi-finger support is available, BTN_TOOL_<name> should be used to report the number of touches active on the trackpad.

For new hardware, INPUT_PROP_POINTER should be set.

Tablets

BTN_TOOL_<name> events must be reported when a stylus or other tool is active on the tablet. ABS_{X,Y} must be reported with the location of the tool. BTN_TOUCH should be used to report when the tool is in contact with the tablet. BTN_{STYLUS,STYLUS2} should be used to report buttons on the tool itself. Any button may be used for buttons on the tablet except BTN_{MOUSE,LEFT}. BTN_{0,1,2,etc} are good generic codes for unlabeled buttons. Do not use meaningful buttons, like BTN_FORWARD, unless the button is labeled for that purpose on the device.

For new hardware, both INPUT_PROP_DIRECT and INPUT_PROP_POINTER should be set.

Multi-touch (MT) Protocol

Introduction

In order to utilize the full power of the new multi-touch and multi-user devices, a way to report detailed data from multiple contacts, i.e., objects in direct contact with the device surface, is needed. This document describes the multi-touch (MT) protocol which allows kernel drivers to report details for an arbitrary number of contacts.

The protocol is divided into two types, depending on the capabilities of the hardware. For devices handling anonymous contacts (type A), the protocol describes how to send the raw data for all contacts to the receiver. For devices capable of tracking identifiable contacts (type B), the protocol describes how to send updates for individual contacts via event slots.

Note:

MT potocol type A is obsolete, all kernel drivers have been converted to use type B.

Protocol Usage

Contact details are sent sequentially as separate packets of ABS_MT events. Only the ABS_MT events are recognized as part of a contact packet. Since these events are ignored by current single-touch (ST) applications, the MT protocol can be implemented on top of the ST protocol in an existing driver.

Drivers for type A devices separate contact packets by calling input_mt_sync() at the end of each packet. This generates a SYN_MT_REPORT event, which instructs the receiver to accept the data for the current contact and prepare to receive another.

Drivers for type B devices separate contact packets by calling input_mt_slot(), with a slot as argument, at the beginning of each packet. This generates an ABS_MT_SLOT event, which instructs the receiver to prepare for updates of the given slot.

All drivers mark the end of a multi-touch transfer by calling the usual input_sync() function. This instructs the receiver to act upon events accumulated since last EV_SYN/SYN_REPORT and prepare to receive a new set of events/packets.

The main difference between the stateless type A protocol and the stateful type B slot protocol lies in the usage of identifiable contacts to reduce the amount of data sent to userspace. The slot protocol requires the use of the ABS_MT_TRACKING_ID, either provided by the hardware or computed from the raw data [5].

For type A devices, the kernel driver should generate an arbitrary enumeration of the full set of anonymous contacts currently on the surface. The order in which the packets appear in the event stream is not important. Event filtering and finger tracking is left to user space [3].

For type B devices, the kernel driver should associate a slot with each identified contact, and use that slot to propagate changes for the contact. Creation, replacement and destruction of contacts is achieved by modifying the ABS_MT_TRACKING_ID of the associated slot. A non-negative tracking id is interpreted as a contact, and the value -1 denotes an unused slot. A tracking id not previously present is considered new, and a tracking id no longer present is considered removed. Since only changes are propagated, the full state of each initiated contact has to reside in the receiving end. Upon receiving an MT event, one simply updates the appropriate attribute of the current slot.

Some devices identify and/or track more contacts than they can report to the driver. A driver for such a device should associate one type B slot with each contact that is reported by the hardware. Whenever the identity of the contact associated with a slot changes, the driver should invalidate that slot by changing its ABS_MT_TRACKING_ID. If the hardware signals that it is tracking more contacts than it is currently reporting, the driver should use a BTN_TOOL_*TAP event to inform userspace of the total

[5] See the section on finger tracking.
[3] The mtdev project: http://bitmath.org/code/mtdev/.

number of contacts being tracked by the hardware at that moment. The driver should do this by explicitly sending the corresponding BTN_TOOL_*TAP event and setting use_count to false when calling input_mt_report_pointer_emulation(). The driver should only advertise as many slots as the hardware can report. Userspace can detect that a driver can report more total contacts than slots by noting that the largest supported BTN_TOOL_*TAP event is larger than the total number of type B slots reported in the absinfo for the ABS_MT_SLOT axis.

The minimum value of the ABS_MT_SLOT axis must be 0.

Protocol Example A

Here is what a minimal event sequence for a two-contact touch would look like for a type A device:

```
ABS_MT_POSITION_X x[0]
ABS_MT_POSITION_Y y[0]
SYN_MT_REPORT
ABS_MT_POSITION_X x[1]
ABS_MT_POSITION_Y y[1]
SYN_MT_REPORT
SYN_REPORT
```

The sequence after moving one of the contacts looks exactly the same; the raw data for all present contacts are sent between every synchronization with SYN_REPORT.

Here is the sequence after lifting the first contact:

```
ABS_MT_POSITION_X x[1]
ABS_MT_POSITION_Y y[1]
SYN_MT_REPORT
SYN_REPORT
```

And here is the sequence after lifting the second contact:

```
SYN_MT_REPORT
SYN_REPORT
```

If the driver reports one of BTN_TOUCH or ABS_PRESSURE in addition to the ABS_MT events, the last SYN_MT_REPORT event may be omitted. Otherwise, the last SYN_REPORT will be dropped by the input core, resulting in no zero-contact event reaching userland.

Protocol Example B

Here is what a minimal event sequence for a two-contact touch would look like for a type B device:

```
ABS_MT_SLOT 0
ABS_MT_TRACKING_ID 45
ABS_MT_POSITION_X x[0]
ABS_MT_POSITION_Y y[0]
ABS_MT_SLOT 1
ABS_MT_TRACKING_ID 46
ABS_MT_POSITION_X x[1]
ABS_MT_POSITION_Y y[1]
SYN_REPORT
```

Here is the sequence after moving contact 45 in the x direction:

```
ABS_MT_SLOT 0
ABS_MT_POSITION_X x[0]
SYN_REPORT
```

Here is the sequence after lifting the contact in slot 0:

```
ABS_MT_TRACKING_ID -1
SYN_REPORT
```

The slot being modified is already 0, so the ABS_MT_SLOT is omitted. The message removes the association of slot 0 with contact 45, thereby destroying contact 45 and freeing slot 0 to be reused for another contact.

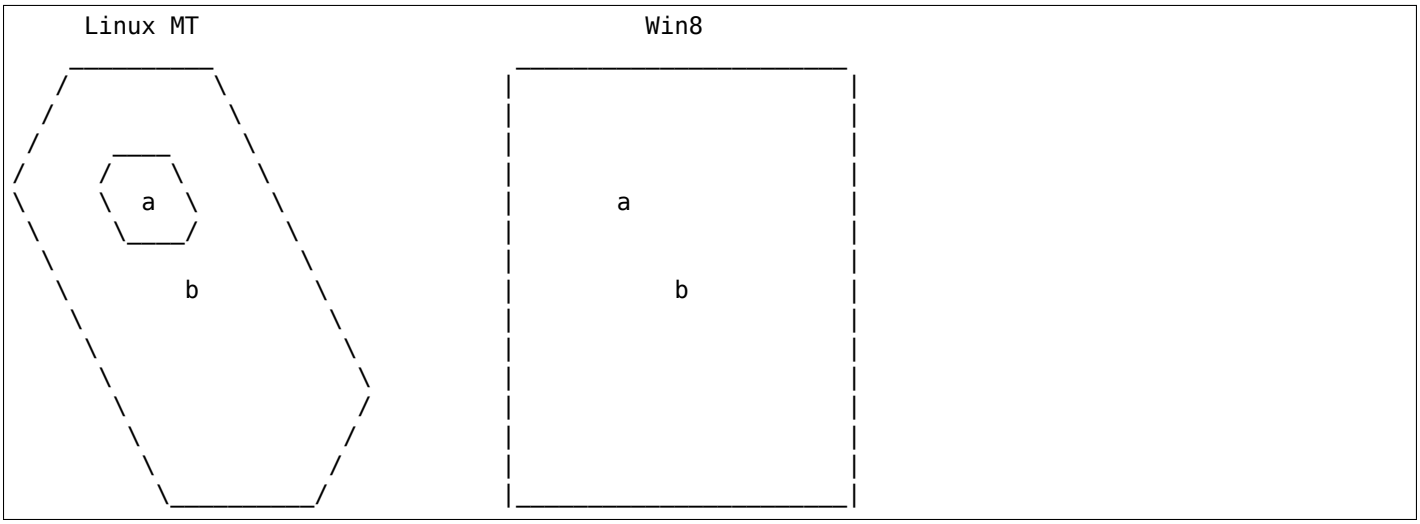
Finally, here is the sequence after lifting the second contact:

```
ABS_MT_SLOT 1
ABS_MT_TRACKING_ID -1
SYN_REPORT
```

Event Usage

A set of ABS_MT events with the desired properties is defined. The events are divided into categories, to allow for partial implementation. The minimum set consists of ABS_MT_POSITION_X and ABS_MT_POSITION_Y, which allows for multiple contacts to be tracked. If the device supports it, the ABS_MT_TOUCH_MAJOR and ABS_MT_WIDTH_MAJOR may be used to provide the size of the contact area and approaching tool, respectively.

The TOUCH and WIDTH parameters have a geometrical interpretation; imagine looking through a window at someone gently holding a finger against the glass. You will see two regions, one inner region consisting of the part of the finger actually touching the glass, and one outer region formed by the perimeter of the finger. The center of the touching region (a) is ABS_MT_POSITION_X/Y and the center of the approaching finger (b) is ABS_MT_TOOL_X/Y. The touch diameter is ABS_MT_TOUCH_MAJOR and the finger diameter is ABS_MT_WIDTH_MAJOR. Now imagine the person pressing the finger harder against the glass. The touch region will increase, and in general, the ratio ABS_MT_TOUCH_MAJOR / ABS_MT_WIDTH_MAJOR, which is always smaller than unity, is related to the contact pressure. For pressure-based devices, ABS_MT_PRESSURE may be used to provide the pressure on the contact area instead. Devices capable of contact hovering can use ABS_MT_DISTANCE to indicate the distance between the contact and the surface.



In addition to the MAJOR parameters, the oval shape of the touch and finger regions can be described by adding the MINOR parameters, such that MAJOR and MINOR are the major and minor axis of an ellipse. The orientation of the touch ellipse can be described with the ORIENTATION parameter, and the direction of the finger ellipse is given by the vector (a - b).

For type A devices, further specification of the touch shape is possible via ABS_MT_BLOB_ID.

The ABS_MT_TOOL_TYPE may be used to specify whether the touching tool is a finger or a pen or something else. Finally, the ABS_MT_TRACKING_ID event may be used to track identified contacts over time [5].

In the type B protocol, ABS_MT_TOOL_TYPE and ABS_MT_TRACKING_ID are implicitly handled by input core; drivers should instead call input_mt_report_slot_state().

Event Semantics

ABS_MT_TOUCH_MAJOR The length of the major axis of the contact. The length should be given in surface units. If the surface has an X times Y resolution, the largest possible value of ABS_MT_TOUCH_MAJOR is sqrt(X^2 + Y^2), the diagonal [4].

ABS_MT_TOUCH_MINOR The length, in surface units, of the minor axis of the contact. If the contact is circular, this event can be omitted [4].

ABS_MT_WIDTH_MAJOR The length, in surface units, of the major axis of the approaching tool. This should be understood as the size of the tool itself. The orientation of the contact and the approaching tool are assumed to be the same [4].

ABS_MT_WIDTH_MINOR The length, in surface units, of the minor axis of the approaching tool. Omit if circular [4].

The above four values can be used to derive additional information about the contact. The ratio ABS_MT_TOUCH_MAJOR / ABS_MT_WIDTH_MAJOR approximates the notion of pressure. The fingers of the hand and the palm all have different characteristic widths.

ABS_MT_PRESSURE The pressure, in arbitrary units, on the contact area. May be used instead of TOUCH and WIDTH for pressure-based devices or any device with a spatial signal intensity distribution.

ABS_MT_DISTANCE The distance, in surface units, between the contact and the surface. Zero distance means the contact is touching the surface. A positive number means the contact is hovering above the surface.

ABS_MT_ORIENTATION The orientation of the touching ellipse. The value should describe a signed quarter of a revolution clockwise around the touch center. The signed value range is arbitrary, but zero should be returned for an ellipse aligned with the Y axis of the surface, a negative value when the ellipse is turned to the left, and a positive value when the ellipse is turned to the right. When completely aligned with the X axis, the range max should be returned.

Touch ellipsis are symmetrical by default. For devices capable of true 360 degree orientation, the reported orientation must exceed the range max to indicate more than a quarter of a revolution. For an upside-down finger, range max * 2 should be returned.

Orientation can be omitted if the touch area is circular, or if the information is not available in the kernel driver. Partial orientation support is possible if the device can distinguish between the two axis, but not (uniquely) any values in between. In such cases, the range of ABS_MT_ORIENTATION should be [0, 1] [4].

ABS_MT_POSITION_X The surface X coordinate of the center of the touching ellipse.

ABS_MT_POSITION_Y The surface Y coordinate of the center of the touching ellipse.

ABS_MT_TOOL_X The surface X coordinate of the center of the approaching tool. Omit if the device cannot distinguish between the intended touch point and the tool itself.

ABS_MT_TOOL_Y The surface Y coordinate of the center of the approaching tool. Omit if the device cannot distinguish between the intended touch point and the tool itself.

The four position values can be used to separate the position of the touch from the position of the tool. If both positions are present, the major tool axis points towards the touch point [1]. Otherwise, the tool axes are aligned with the touch axes.

[4] See the section on event computation.
[1] Also, the difference (TOOL_X - POSITION_X) can be used to model tilt.

ABS_MT_TOOL_TYPE The type of approaching tool. A lot of kernel drivers cannot distinguish between different tool types, such as a finger or a pen. In such cases, the event should be omitted. The protocol currently supports MT_TOOL_FINGER, MT_TOOL_PEN, and MT_TOOL_PALM [2]. For type B devices, this event is handled by input core; drivers should instead use input_mt_report_slot_state(). A contact's ABS_MT_TOOL_TYPE may change over time while still touching the device, because the firmware may not be able to determine which tool is being used when it first appears.

ABS_MT_BLOB_ID The BLOB_ID groups several packets together into one arbitrarily shaped contact. The sequence of points forms a polygon which defines the shape of the contact. This is a low-level anonymous grouping for type A devices, and should not be confused with the high-level trackingID [5]. Most type A devices do not have blob capability, so drivers can safely omit this event.

ABS_MT_TRACKING_ID The TRACKING_ID identifies an initiated contact throughout its life cycle [5]. The value range of the TRACKING_ID should be large enough to ensure unique identification of a contact maintained over an extended period of time. For type B devices, this event is handled by input core; drivers should instead use input_mt_report_slot_state().

Event Computation

The flora of different hardware unavoidably leads to some devices fitting better to the MT protocol than others. To simplify and unify the mapping, this section gives recipes for how to compute certain events.

For devices reporting contacts as rectangular shapes, signed orientation cannot be obtained. Assuming X and Y are the lengths of the sides of the touching rectangle, here is a simple formula that retains the most information possible:

```
ABS_MT_TOUCH_MAJOR := max(X, Y)
ABS_MT_TOUCH_MINOR := min(X, Y)
ABS_MT_ORIENTATION := bool(X > Y)
```

The range of ABS_MT_ORIENTATION should be set to [0, 1], to indicate that the device can distinguish between a finger along the Y axis (0) and a finger along the X axis (1).

For win8 devices with both T and C coordinates, the position mapping is:

```
ABS_MT_POSITION_X := T_X
ABS_MT_POSITION_Y := T_Y
ABS_MT_TOOL_X := C_X
ABS_MT_TOOL_Y := C_Y
```

Unfortunately, there is not enough information to specify both the touching ellipse and the tool ellipse, so one has to resort to approximations. One simple scheme, which is compatible with earlier usage, is:

```
ABS_MT_TOUCH_MAJOR := min(X, Y)
ABS_MT_TOUCH_MINOR := <not used>
ABS_MT_ORIENTATION := <not used>
ABS_MT_WIDTH_MAJOR := min(X, Y) + distance(T, C)
ABS_MT_WIDTH_MINOR := min(X, Y)
```

Rationale: We have no information about the orientation of the touching ellipse, so approximate it with an inscribed circle instead. The tool ellipse should align with the vector (T - C), so the diameter must increase with distance(T, C). Finally, assume that the touch diameter is equal to the tool thickness, and we arrive at the formulas above.

Finger Tracking

The process of finger tracking, i.e., to assign a unique trackingID to each initiated contact on the surface, is a Euclidian Bipartite Matching problem. At each event synchronization, the set of actual contacts is

[2] The list can of course be extended.

matched to the set of contacts from the previous synchronization. A full implementation can be found in [3].

Gestures

In the specific application of creating gesture events, the TOUCH and WIDTH parameters can be used to, e.g., approximate finger pressure or distinguish between index finger and thumb. With the addition of the MINOR parameters, one can also distinguish between a sweeping finger and a pointing finger, and with ORIENTATION, one can detect twisting of fingers.

Notes

In order to stay compatible with existing applications, the data reported in a finger packet must not be recognized as single-touch events.

For type A devices, all finger data bypasses input filtering, since subsequent events of the same type refer to different fingers.

Linux Gamepad Specification

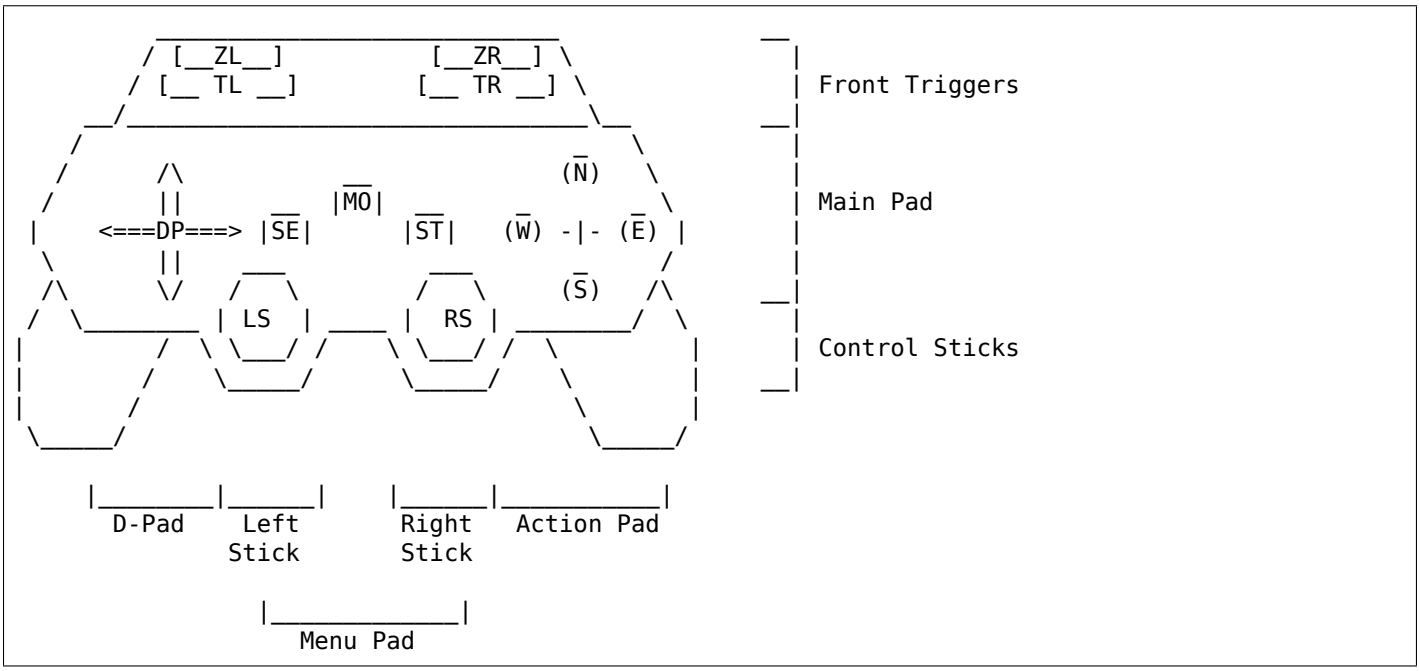
Author 2013 by David Herrmann <dh.herrmann@gmail.com>

Introduction

Linux provides many different input drivers for gamepad hardware. To avoid having user-space deal with different button-mappings for each gamepad, this document defines how gamepads are supposed to report their data.

Geometry

As “gamepad” we define devices which roughly look like this:



Most gamepads have the following features:

- Action-Pad 4 buttons in diamonds-shape (on the right side). The buttons are differently labeled on most devices so we define them as NORTH, SOUTH, WEST and EAST.
- D-Pad (Direction-pad) 4 buttons (on the left side) that point up, down, left and right.
- Menu-Pad Different constellations, but most-times 2 buttons: SELECT - START Furthermore, many gamepads have a fancy branded button that is used as special system-button. It often looks different to the other buttons and is used to pop up system-menus or system-settings.
- Analog-Sticks Analog-sticks provide freely moveable sticks to control directions. Not all devices have both or any, but they are present at most times. Analog-sticks may also provide a digital button if you press them.
- Triggers Triggers are located on the upper-side of the pad in vertical direction. Not all devices provide them, but the upper buttons are normally named Left- and Right-Triggers, the lower buttons Z-Left and Z-Right.
- Rumble Many devices provide force-feedback features. But are mostly just simple rumble motors.

Detection

All gamepads that follow the protocol described here map BTN_GAMEPAD. This is an alias for BTN_SOUTH/BTN_A. It can be used to identify a gamepad as such. However, not all gamepads provide all features, so you need to test for all features that you need, first. How each feature is mapped is described below.

Legacy drivers often don't comply to these rules. As we cannot change them for backwards-compatibility reasons, you need to provide fixup mappings in user-space yourself. Some of them might also provide module-options that change the mappings so you can advise users to set these.

All new gamepads are supposed to comply with this mapping. Please report any bugs, if they don't.

There are a lot of less-featured/less-powerful devices out there, which re-use the buttons from this protocol. However, they try to do this in a compatible fashion. For example, the "Nintendo Wii Nunchuk" provides two trigger buttons and one analog stick. It reports them as if it were a gamepad with only one analog stick and two trigger buttons on the right side. But that means, that if you only support "real" gamepads, you must test devices for _all_ reported events that you need. Otherwise, you will also get devices that report a small subset of the events.

No other devices, that do not look/feel like a gamepad, shall report these events.

Events

Gamepads report the following events:

- Action-Pad:

  Every gamepad device has at least 2 action buttons. This means, that every device reports BTN_SOUTH (which BTN_GAMEPAD is an alias for). Regardless of the labels on the buttons, the codes are sent according to the physical position of the buttons.

  Please note that 2- and 3-button pads are fairly rare and old. You might want to filter gamepads that do not report all four.

  - 2-Button Pad:

    If only 2 action-buttons are present, they are reported as BTN_SOUTH and BTN_EAST. For vertical layouts, the upper button is BTN_EAST. For horizontal layouts, the button more on the right is BTN_EAST.

- 3-Button Pad:

  If only 3 action-buttons are present, they are reported as (from left to right): BTN_WEST, BTN_SOUTH, BTN_EAST If the buttons are aligned perfectly vertically, they are reported as (from top down): BTN_WEST, BTN_SOUTH, BTN_EAST

- 4-Button Pad:

  If all 4 action-buttons are present, they can be aligned in two different formations. If diamond-shaped, they are reported as BTN_NORTH, BTN_WEST, BTN_SOUTH, BTN_EAST according to their physical location. If rectangular-shaped, the upper-left button is BTN_NORTH, lower-left is BTN_WEST, lower-right is BTN_SOUTH and upper-right is BTN_EAST.

- D-Pad:

  Every gamepad provides a D-Pad with four directions: Up, Down, Left, Right Some of these are available as digital buttons, some as analog buttons. Some may even report both. The kernel does not convert between these so applications should support both and choose what is more appropriate if both are reported.

  - Digital buttons are reported as:

    BTN_DPAD_*

  - Analog buttons are reported as:

    ABS_HAT0X and ABS_HAT0Y

  (for ABS values negative is left/up, positive is right/down)

- Analog-Sticks:

  The left analog-stick is reported as ABS_X, ABS_Y. The right analog stick is reported as ABS_RX, ABS_RY. Zero, one or two sticks may be present. If analog-sticks provide digital buttons, they are mapped accordingly as BTN_THUMBL (first/left) and BTN_THUMBR (second/right).

  (for ABS values negative is left/up, positive is right/down)

- Triggers:

  Trigger buttons can be available as digital or analog buttons or both. User- space must correctly deal with any situation and choose the most appropriate mode.

  Upper trigger buttons are reported as BTN_TR or ABS_HAT1X (right) and BTN_TL or ABS_HAT1Y (left). Lower trigger buttons are reported as BTN_TR2 or ABS_HAT2X (right/ZR) and BTN_TL2 or ABS_HAT2Y (left/ZL).

  If only one trigger-button combination is present (upper+lower), they are reported as “right” triggers (BTN_TR/ABS_HAT1X).

  (ABS trigger values start at 0, pressure is reported as positive values)

- Menu-Pad:

  Menu buttons are always digital and are mapped according to their location instead of their labels. That is:

  - 1-button Pad:

    Mapped as BTN_START

  - 2-button Pad:

    Left button mapped as BTN_SELECT, right button mapped as BTN_START

  Many pads also have a third button which is branded or has a special symbol and meaning. Such buttons are mapped as BTN_MODE. Examples are the Nintendo “HOME” button, the XBox “X”-button or Sony “PS” button.

- Rumble:

  Rumble is advertised as FF_RUMBLE.

Force feedback for Linux

Author Johann Deneux <johann.deneux@gmail.com> on 2001/04/22.

Updated Anssi Hannula <anssi.hannula@gmail.com> on 2006/04/09.

Introduction

This document describes how to use force feedback devices under Linux. The goal is not to support these devices as if they were simple input-only devices (as it is already the case), but to really enable the rendering of force effects. This document only describes the force feedback part of the Linux input interface. Please read joystick.txt and input.txt before reading further this document.

Instructions to the user

To enable force feedback, you have to:

1. have your kernel configured with evdev and a driver that supports your device.
2. make sure evdev module is loaded and /dev/input/event* device files are created.

Before you start, let me WARN you that some devices shake violently during the initialisation phase. This happens for example with my "AVB Top Shot Pegasus". To stop this annoying behaviour, move you joystick to its limits. Anyway, you should keep a hand on your device, in order to avoid it to break down if something goes wrong.

If you have a serial iforce device, you need to start inputattach. See joystick.txt for details.

Does it work ?

There is an utility called fftest that will allow you to test the driver:

```
% fftest /dev/input/eventXX
```

Instructions to the developer

All interactions are done using the event API. That is, you can use ioctl() and write() on /dev/input/eventXX. This information is subject to change.

Querying device capabilities

```
#include <linux/input.h>
#include <sys/ioctl.h>

#define BITS_TO_LONGS(x) \
        (((x) + 8 * sizeof (unsigned long) - 1) / (8 * sizeof (unsigned long)))
unsigned long features[BITS_TO_LONGS(FF_CNT)];
int ioctl(int file_descriptor, int request, unsigned long *features);
```

"request" must be EVIOCGBIT(EV_FF, size of features array in bytes)

Returns the features supported by the device. features is a bitfield with the following bits:

- FF_CONSTANT can render constant force effects
- FF_PERIODIC can render periodic effects with the following waveforms:

  - FF_SQUARE square waveform
  - FF_TRIANGLE triangle waveform
  - FF_SINE sine waveform
  - FF_SAW_UP sawtooth up waveform
  - FF_SAW_DOWN sawtooth down waveform
  - FF_CUSTOM custom waveform
- FF_RAMP can render ramp effects
- FF_SPRING can simulate the presence of a spring
- FF_FRICTION can simulate friction
- FF_DAMPER can simulate damper effects
- FF_RUMBLE rumble effects
- FF_INERTIA can simulate inertia
- FF_GAIN gain is adjustable
- FF_AUTOCENTER autocenter is adjustable

Note:

- *In most cases you should use FF_PERIODIC instead of FF_RUMBLE. All devices that support FF_RUMBLE support FF_PERIODIC (square, triangle, sine) and the other way around.*
- *The exact syntax FF_CUSTOM is undefined for the time being as no driver supports it yet.*

```
int ioctl(int fd, EVIOCGEFFECTS, int *n);
```

Returns the number of effects the device can keep in its memory.

Uploading effects to the device

```
#include <linux/input.h>
#include <sys/ioctl.h>

int ioctl(int file_descriptor, int request, struct ff_effect *effect);
```

“request” must be EVIOCSFF.

“effect” points to a structure describing the effect to upload. The effect is uploaded, but not played. The content of effect may be modified. In particular, its field “id” is set to the unique id assigned by the driver. This data is required for performing some operations (removing an effect, controlling the playback). This if field must be set to -1 by the user in order to tell the driver to allocate a new effect.

Effects are file descriptor specific.

See <uapi/linux/input.h> for a description of the ff_effect struct. You should also find help in a few sketches, contained in files shape.svg and interactive.svg:

Removing an effect from the device

```
int ioctl(int fd, EVIOCRMFF, effect.id);
```

This makes room for new effects in the device’s memory. Note that this also stops the effect if it was playing.

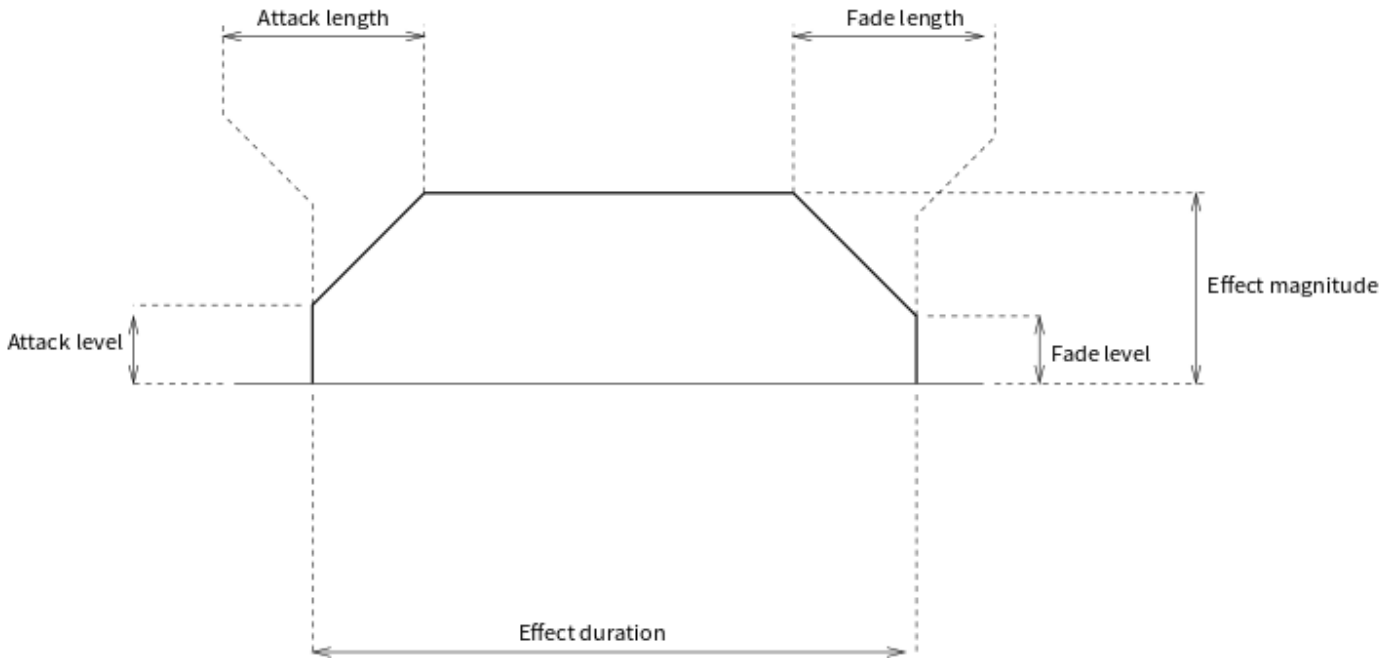


Fig. 1.1: Shape

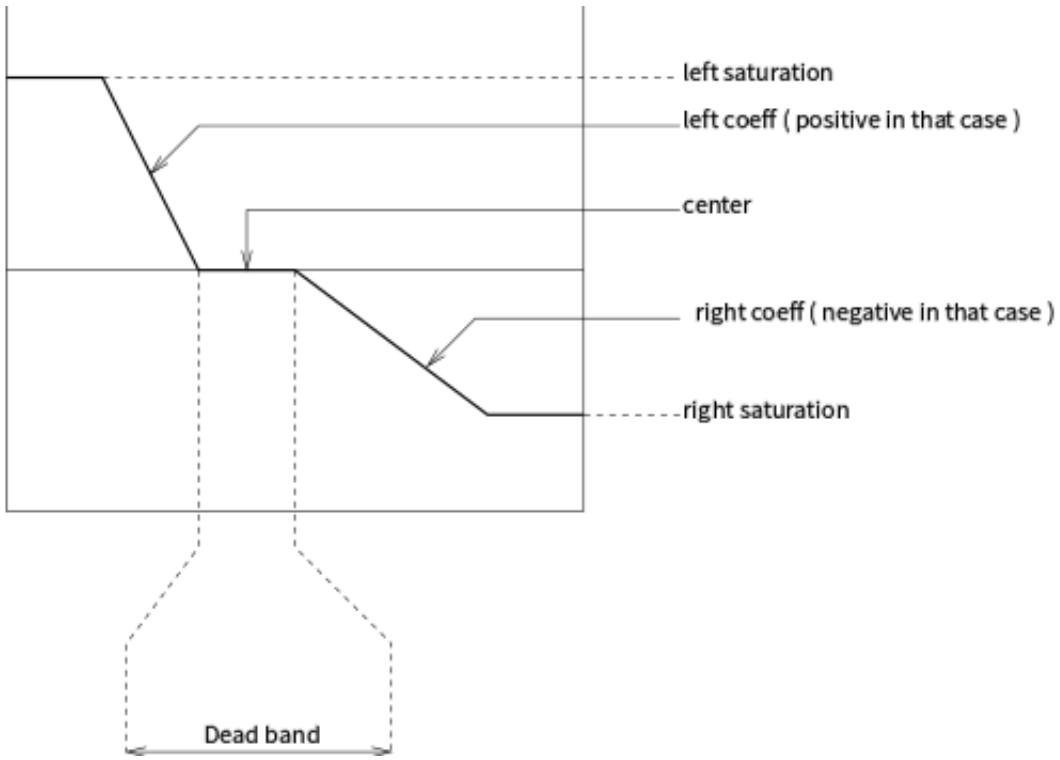


Fig. 1.2: Interactive

Controlling the playback of effects

Control of playing is done with write(). Below is an example:

```
 #include <linux/input.h>
 #include <unistd.h>

     struct input_event play;
     struct input_event stop;
     struct ff_effect effect;
     int fd;
...
     fd = open("/dev/input/eventXX", O_RDWR);
...
     /* Play three times */
     play.type = EV_FF;
     play.code = effect.id;
     play.value = 3;

     write(fd, (const void*) &play, sizeof(play));
...
     /* Stop an effect */
     stop.type = EV_FF;
     stop.code = effect.id;
     stop.value = 0;

     write(fd, (const void*) &play, sizeof(stop));
```

Setting the gain

Not all devices have the same strength. Therefore, users should set a gain factor depending on how strong they want effects to be. This setting is persistent across access to the driver.

```
/* Set the gain of the device
int gain;           /* between 0 and 100 */
struct input_event ie;      /* structure used to communicate with the driver */

ie.type = EV_FF;
ie.code = FF_GAIN;
ie.value = 0xFFFFUL * gain / 100;

if (write(fd, &ie, sizeof(ie)) == -1)
    perror("set gain");
```

Enabling/Disabling autocenter

The autocenter feature quite disturbs the rendering of effects in my opinion, and I think it should be an effect, which computation depends on the game type. But you can enable it if you want.

```
int autocenter;             /* between 0 and 100 */
struct input_event ie;

ie.type = EV_FF;
ie.code = FF_AUTOCENTER;
ie.value = 0xFFFFUL * autocenter / 100;

if (write(fd, &ie, sizeof(ie)) == -1)
    perror("set auto-center");
```

A value of 0 means “no auto-center”.

Dynamic update of an effect

Proceed as if you wanted to upload a new effect, except that instead of setting the id field to -1, you set it to the wanted effect id. Normally, the effect is not stopped and restarted. However, depending on the type of device, not all parameters can be dynamically updated. For example, the direction of an effect cannot be updated with iforce devices. In this case, the driver stops the effect, up-load it, and restart it.

Therefore it is recommended to dynamically change direction while the effect is playing only when it is ok to restart the effect with a replay count of 1.

Information about the status of effects

Every time the status of an effect is changed, an event is sent. The values and meanings of the fields of the event are as follows:

```
struct input_event {
/* When the status of the effect changed */
        struct timeval time;

/* Set to EV_FF_STATUS */
        unsigned short type;

/* Contains the id of the effect */
        unsigned short code;

/* Indicates the status */
        unsigned int value;
};

FF_STATUS_STOPPED   The effect stopped playing
FF_STATUS_PLAYING   The effect started to play
```

Note:

- *Status feedback is only supported by iforce driver. If you have a really good reason to use this, please contact linux-joystick@atrey.karlin.mff.cuni.cz or anssi.hannula@gmail.com so that support for it can be added to the rest of the drivers.*

Linux Joystick support

Table of Contents

Introduction

The joystick driver for Linux provides support for a variety of joysticks and similar devices. It is based on a larger project aiming to support all input devices in Linux.

The mailing list for the project is:

linux-input@vger.kernel.org

send “subscribe linux-input” to majordomo@vger.kernel.org to subscribe to it.

Usage

For basic usage you just choose the right options in kernel config and you should be set.

Utilities

For testing and other purposes (for example serial devices), there is a set of utilities, such as `jstest`, `jscal`, and `evtest`, usually packaged as `joystick`, `input-utils`, `evtest`, and so on.

`inputattach` utility is required if your joystick is connected to a serial port.

Device nodes

For applications to be able to use the joysticks, device nodes should be created in /dev. Normally it is done automatically by the system, but it can also be done by hand:

```
cd /dev
rm js*
mkdir input
mknod input/js0 c 13 0
mknod input/js1 c 13 1
mknod input/js2 c 13 2
mknod input/js3 c 13 3
ln -s input/js0 js0
ln -s input/js1 js1
ln -s input/js2 js2
ln -s input/js3 js3
```

For testing with inpututils it's also convenient to create these:

```
mknod input/event0 c 13 64
mknod input/event1 c 13 65
mknod input/event2 c 13 66
mknod input/event3 c 13 67
```

Modules needed

For all joystick drivers to function, you'll need the userland interface module in kernel, either loaded or compiled in:

```
modprobe joydev
```

For gameport joysticks, you'll have to load the gameport driver as well:

```
modprobe ns558
```

And for serial port joysticks, you'll need the serial input line discipline module loaded and the inputattach utility started:

```
modprobe serport
inputattach -xxx /dev/tts/X &
```

In addition to that, you'll need the joystick driver module itself, most usually you'll have an analog joystick:

```
modprobe analog
```

For automatic module loading, something like this might work - tailor to your needs:

```
alias tty-ldisc-2 serport
alias char-major-13 input
above input joydev ns558 analog
options analog map=gamepad,none,2btn
```

Verifying that it works

For testing the joystick driver functionality, there is the jstest program in the utilities package. You run it by typing:

```
jstest /dev/input/js0
```

And it should show a line with the joystick values, which update as you move the stick, and press its buttons. The axes should all be zero when the joystick is in the center position. They should not jitter by themselves to other close values, and they also should be steady in any other position of the stick. They should have the full range from -32767 to 32767. If all this is met, then it's all fine, and you can play the games. :)

If it's not, then there might be a problem. Try to calibrate the joystick, and if it still doesn't work, read the drivers section of this file, the troubleshooting section, and the FAQ.

Calibration

For most joysticks you won't need any manual calibration, since the joystick should be autocalibrated by the driver automagically. However, with some analog joysticks, that either do not use linear resistors, or if you want better precision, you can use the jscal program:

```
jscal -c /dev/input/js0
```

included in the joystick package to set better correction coefficients than what the driver would choose itself.

After calibrating the joystick you can verify if you like the new calibration using the jstest command, and if you do, you then can save the correction coefficients into a file:

```
jscal -p /dev/input/js0 > /etc/joystick.cal
```

And add a line to your rc script executing that file:

```
source /etc/joystick.cal
```

This way, after the next reboot your joystick will remain calibrated. You can also add the `jscal -p` line to your shutdown script.

HW specific driver information

In this section each of the separate hardware specific drivers is described.

Analog joysticks

The analog.c uses the standard analog inputs of the gameport, and thus supports all standard joysticks and gamepads. It uses a very advanced routine for this, allowing for data precision that can't be found on any other system.

It also supports extensions like additional hats and buttons compatible with CH Flightstick Pro, ThrustMaster FCS or 6 and 8 button gamepads. Saitek Cyborg 'digital' joysticks are also supported by this driver, because they're basically souped up CHF sticks.

However the only types that can be autodetected are:

- 2-axis, 4-button joystick
- 3-axis, 4-button joystick
- 4-axis, 4-button joystick
- Saitek Cyborg 'digital' joysticks

For other joystick types (more/less axes, hats, and buttons) support you'll need to specify the types either on the kernel command line or on the module command line, when inserting analog into the kernel. The parameters are:

```
analog.map=<type1>,<type2>,<type3>,....
```

'type' is type of the joystick from the table below, defining joysticks present on gameports in the system, starting with gameport0, second 'type' entry defining joystick on gameport1 and so on.

Type	Meaning
none	No analog joystick on that port
auto	Autodetect joystick
2btn	2-button n-axis joystick
y-joy	Two 2-button 2-axis joysticks on an Y-cable
y-pad	Two 2-button 2-axis gamepads on an Y-cable
fcs	Thrustmaster FCS compatible joystick
chf	Joystick with a CH Flightstick compatible hat
fullchf	CH Flightstick compatible with two hats and 6 buttons
gamepad	4/6-button n-axis gamepad
gamepad8	8-button 2-axis gamepad

In case your joystick doesn't fit in any of the above categories, you can specify the type as a number by combining the bits in the table below. This is not recommended unless you really know what are you doing. It's not dangerous, but not simple either.

Bit	Meaning
0	Axis X1
1	Axis Y1
2	Axis X2
3	Axis Y2
4	Button A
5	Button B
6	Button C
7	Button D
8	CHF Buttons X and Y
9	CHF Hat 1
10	CHF Hat 2
11	FCS Hat
12	Pad Button X
13	Pad Button Y
14	Pad Button U
15	Pad Button V
16	Saitek F1-F4 Buttons
17	Saitek Digital Mode
19	GamePad
20	Joy2 Axis X1
21	Joy2 Axis Y1
22	Joy2 Axis X2
23	Joy2 Axis Y2
24	Joy2 Button A
25	Joy2 Button B
26	Joy2 Button C
27	Joy2 Button D
31	Joy2 GamePad

Microsoft SideWinder joysticks

Microsoft 'Digital Overdrive' protocol is supported by the sidewinder.c module. All currently supported joysticks:

- Microsoft SideWinder 3D Pro
- Microsoft SideWinder Force Feedback Pro
- Microsoft SideWinder Force Feedback Wheel
- Microsoft SideWinder FreeStyle Pro
- Microsoft SideWinder GamePad (up to four, chained)
- Microsoft SideWinder Precision Pro
- Microsoft SideWinder Precision Pro USB

are autodetected, and thus no module parameters are needed.

There is one caveat with the 3D Pro. There are 9 buttons reported, although the joystick has only 8. The 9th button is the mode switch on the rear side of the joystick. However, moving it, you'll reset the joystick, and make it unresponsive for about a one third of a second. Furthermore, the joystick will also re-center itself, taking the position it was in during this time as a new center position. Use it if you want, but think first.

The SideWinder Standard is not a digital joystick, and thus is supported by the analog driver described above.

Logitech ADI devices

Logitech ADI protocol is supported by the adi.c module. It should support any Logitech device using this protocol. This includes, but is not limited to:

- Logitech CyberMan 2
- Logitech ThunderPad Digital
- Logitech WingMan Extreme Digital
- Logitech WingMan Formula
- Logitech WingMan Interceptor
- Logitech WingMan GamePad
- Logitech WingMan GamePad USB
- Logitech WingMan GamePad Extreme
- Logitech WingMan Extreme Digital 3D

ADI devices are autodetected, and the driver supports up to two (any combination of) devices on a single gameport, using an Y-cable or chained together.

Logitech WingMan Joystick, Logitech WingMan Attack, Logitech WingMan Extreme and Logitech WingMan ThunderPad are not digital joysticks and are handled by the analog driver described above. Logitech WingMan Warrior and Logitech Magellan are supported by serial drivers described below. Logitech WingMan Force and Logitech WingMan Formula Force are supported by the I-Force driver described below. Logitech CyberMan is not supported yet.

Gravis GrIP

Gravis GrIP protocol is supported by the grip.c module. It currently supports:

- Gravis GamePad Pro
- Gravis BlackHawk Digital
- Gravis Xterminator
- Gravis Xterminator DualControl

All these devices are autodetected, and you can even use any combination of up to two of these pads either chained together or using an Y-cable on a single gameport.

GrIP MultiPort isn't supported yet. Gravis Stinger is a serial device and is supported by the stinger driver. Other Gravis joysticks are supported by the analog driver.

FPGaming A3D and MadCatz A3D

The Assassin 3D protocol created by FPGaming, is used both by FPGaming themselves and is licensed to MadCatz. A3D devices are supported by the a3d.c module. It currently supports:

- FPGaming Assassin 3D
- MadCatz Panther
- MadCatz Panther XL

All these devices are autodetected. Because the Assassin 3D and the Panther allow connecting analog joysticks to them, you'll need to load the analog driver as well to handle the attached joysticks.

The trackball should work with USB mousedev module as a normal mouse. See the USB documentation for how to setup an USB mouse.

ThrustMaster DirectConnect (BSP)

The TM DirectConnect (BSP) protocol is supported by the tmdc.c module. This includes, but is not limited to:

- ThrustMaster Millennium 3D Interceptor
- ThrustMaster 3D Rage Pad
- ThrustMaster Fusion Digital Game Pad

Devices not directly supported, but hopefully working are:

- ThrustMaster FragMaster
- ThrustMaster Attack Throttle

If you have one of these, contact me.

TMDC devices are autodetected, and thus no parameters to the module are needed. Up to two TMDC devices can be connected to one gameport, using an Y-cable.

Creative Labs Blaster

The Blaster protocol is supported by the cobra.c module. It supports only the:

- Creative Blaster GamePad Cobra

Up to two of these can be used on a single gameport, using an Y-cable.

Genius Digital joysticks

The Genius digitally communicating joysticks are supported by the gf2k.c module. This includes:

- Genius Flight2000 F-23 joystick
- Genius Flight2000 F-31 joystick
- Genius G-09D gamepad

Other Genius digital joysticks are not supported yet, but support can be added fairly easily.

InterAct Digital joysticks

The InterAct digitally communicating joysticks are supported by the interact.c module. This includes:

- InterAct HammerHead/FX gamepad
- InterAct ProPad8 gamepad

Other InterAct digital joysticks are not supported yet, but support can be added fairly easily.

PDPI Lightning 4 gamecards

PDPI Lightning 4 gamecards are supported by the lightning.c module. Once the module is loaded, the analog driver can be used to handle the joysticks. Digitally communicating joystick will work only on port 0, while using Y-cables, you can connect up to 8 analog joysticks to a single L4 card, 16 in case you have two in your system.

Trident 4DWave / Aureal Vortex

Soundcards with a Trident 4DWave DX/NX or Aureal Vortex/Vortex2 chipsets provide an "Enhanced Game Port" mode where the soundcard handles polling the joystick. This mode is supported by the pcigame.c module. Once loaded the analog driver can use the enhanced features of these gameports..

Crystal SoundFusion

Soundcards with Crystal SoundFusion chipsets provide an "Enhanced Game Port", much like the 4DWave or Vortex above. This, and also the normal mode for the port of the SoundFusion is supported by the cs461x.c module.

SoundBlaster Live!

The Live! has a special PCI gameport, which, although it doesn't provide any "Enhanced" stuff like 4DWave and friends, is quite a bit faster than its ISA counterparts. It also requires special support, hence the emu10k1-gp.c module for it instead of the normal ns558.c one.

SoundBlaster 64 and 128 - ES1370 and ES1371, ESS Solo1 and S3 SonicVibes

These PCI soundcards have specific gameports. They are handled by the sound drivers themselves. Make sure you select gameport support in the joystick menu and sound card support in the sound menu for your appropriate card.

Amiga

Amiga joysticks, connected to an Amiga, are supported by the amijoy.c driver. Since they can't be autodetected, the driver has a command line:

```
amijoy.map=<a>,<b>
```

a and b define the joysticks connected to the JOY0DAT and JOY1DAT ports of the Amiga.

Value	Joystick type
0	None
1	1-button digital joystick

No more joystick types are supported now, but that should change in the future if I get an Amiga in the reach of my fingers.

Game console and 8-bit pads and joysticks

These pads and joysticks are not designed for PCs and other computers Linux runs on, and usually require a special connector for attaching them through a parallel port.

See *Parallel Port Joystick Drivers* for more info.

SpaceTec/LabTec devices

SpaceTec serial devices communicate using the SpaceWare protocol. It is supported by the spaceorb.c and spaceball.c drivers. The devices currently supported by spaceorb.c are:

- SpaceTec SpaceBall Avenger
- SpaceTec SpaceOrb 360

Devices currently supported by spaceball.c are:

- SpaceTec SpaceBall 4000 FLX

In addition to having the spaceorb/spaceball and serport modules in the kernel, you also need to attach a serial port to it. to do that, run the inputattach program:

```
inputattach --spaceorb /dev/tts/x &
```

or:

```
inputattach --spaceball /dev/tts/x &
```

where /dev/tts/x is the serial port which the device is connected to. After doing this, the device will be reported and will start working.

There is one caveat with the SpaceOrb. The button #6, the on the bottom side of the orb, although reported as an ordinary button, causes internal recentering of the spaceorb, moving the zero point to the position in which the ball is at the moment of pressing the button. So, think first before you bind it to some other function.

SpaceTec SpaceBall 2003 FLX and 3003 FLX are not supported yet.

Logitech SWIFT devices

The SWIFT serial protocol is supported by the warrior.c module. It currently supports only the:

- Logitech WingMan Warrior

but in the future, Logitech CyberMan (the original one, not CM2) could be supported as well. To use the module, you need to run inputattach after you insert/compile the module into your kernel:

```
inputattach --warrior /dev/tts/x &
```

/dev/tts/x is the serial port your Warrior is attached to.

Magellan / Space Mouse

The Magellan (or Space Mouse), manufactured by LogiCad3d (formerly Space Systems), for many other companies (Logitech, HP, ...) is supported by the joy-magellan module. It currently supports only the:

- Magellan 3D
- Space Mouse

models, the additional buttons on the 'Plus' versions are not supported yet.

To use it, you need to attach the serial port to the driver using the:

```
inputattach --magellan /dev/tts/x &
```

command. After that the Magellan will be detected, initialized, will beep, and the /dev/input/jsX device should become usable.

I-Force devices

All I-Force devices are supported by the iforce module. This includes:

- AVB Mag Turbo Force
- AVB Top Shot Pegasus
- AVB Top Shot Force Feedback Racing Wheel
- Logitech WingMan Force

- Logitech WingMan Force Wheel
- Guillemot Race Leader Force Feedback
- Guillemot Force Feedback Racing Wheel
- Thrustmaster Motor Sport GT

To use it, you need to attach the serial port to the driver using the:

```
inputattach --iforce /dev/tts/x &
```

command. After that the I-Force device will be detected, and the /dev/input/jsX device should become usable.

In case you're using the device via the USB port, the inputattach command isn't needed.

The I-Force driver now supports force feedback via the event interface.

Please note that Logitech WingMan 3D devices are _not_ supported by this module, rather by hid. Force feedback is not supported for those devices. Logitech gamepads are also hid devices.

Gravis Stinger gamepad

The Gravis Stinger serial port gamepad, designed for use with laptop computers, is supported by the stinger.c module. To use it, attach the serial port to the driver using:

```
inputattach --stinger /dev/tty/x &
```

where x is the number of the serial port.

Troubleshooting

There is quite a high probability that you run into some problems. For testing whether the driver works, if in doubt, use the jstest utility in some of its modes. The most useful modes are "normal" - for the 1.x interface, and "old" for the "0.x" interface. You run it by typing:

```
jstest --normal /dev/input/js0
jstest --old    /dev/input/js0
```

Additionally you can do a test with the evtest utility:

```
evtest /dev/input/event0
```

Oh, and read the FAQ! :)

FAQ

Q Running 'jstest /dev/input/js0' results in "File not found" error. What's the cause?

A The device files don't exist. Create them (see section 2.2).

Q Is it possible to connect my old Atari/Commodore/Amiga/console joystick or pad that uses a 9-pin D-type cannon connector to the serial port of my PC?

A Yes, it is possible, but it'll burn your serial port or the pad. It won't work, of course.

Q My joystick doesn't work with Quake / Quake 2. What's the cause?

A Quake / Quake 2 don't support joystick. Use joy2key to simulate keypresses for them.

Programming Interface

Author Ragnar Hojland Espinosa <ragnar@macula.net> - 7 Aug 1998

Introduction

Important:

This document describes legacy js interface. Newer clients are encouraged to switch to the generic event (evdev) interface.

The 1.0 driver uses a new, event based approach to the joystick driver. Instead of the user program polling for the joystick values, the joystick driver now reports only any changes of its state. See joystick-api.txt, joystick.h and jstest.c included in the joystick package for more information. The joystick device can be used in either blocking or nonblocking mode, and supports select() calls.

For backward compatibility the old (v0.x) interface is still included. Any call to the joystick driver using the old interface will return values that are compatible to the old interface. This interface is still limited to 2 axes, and applications using it usually decode only 2 buttons, although the driver provides up to 32.

Initialization

Open the joystick device following the usual semantics (that is, with open). Since the driver now reports events instead of polling for changes, immediately after the open it will issue a series of synthetic events (JS_EVENT_INIT) that you can read to obtain the initial state of the joystick.

By default, the device is opened in blocking mode:

```
int fd = open ("/dev/input/js0", O_RDONLY);
```

Event Reading

```
struct js_event e;
read (fd, &e, sizeof(e));
```

where js_event is defined as:

```
struct js_event {
        __u32 time;     /* event timestamp in milliseconds */
        __s16 value;    /* value */
        __u8 type;      /* event type */
        __u8 number;    /* axis/button number */
};
```

If the read is successful, it will return sizeof(e), unless you wanted to read more than one event per read as described in section 3.1.

js_event.type

The possible values of type are:

```
#define JS_EVENT_BUTTON         0x01    /* button pressed/released */
#define JS_EVENT_AXIS           0x02    /* joystick moved */
#define JS_EVENT_INIT           0x80    /* initial state of device */
```

As mentioned above, the driver will issue synthetic JS_EVENT_INIT ORed events on open. That is, if it's issuing a INIT BUTTON event, the current type value will be:

```
int type = JS_EVENT_BUTTON | JS_EVENT_INIT;     /* 0x81 */
```

If you choose not to differentiate between synthetic or real events you can turn off the JS_EVENT_INIT bits:

```
type &= ~JS_EVENT_INIT;                         /* 0x01 */
```

js_event.number

The values of `number` correspond to the axis or button that generated the event. Note that they carry separate numeration (that is, you have both an axis 0 and a button 0). Generally,

Axis	number
1st Axis X	0
1st Axis Y	1
2nd Axis X	2
2nd Axis Y	3
...and so on	

Hats vary from one joystick type to another. Some can be moved in 8 directions, some only in 4, The driver, however, always reports a hat as two independent axis, even if the hardware doesn't allow independent movement.

js_event.value

For an axis, `value` is a signed integer between -32767 and +32767 representing the position of the joystick along that axis. If you don't read a 0 when the joystick is dead, or if it doesn't span the full range, you should recalibrate it (with, for example, jscal).

For a button, `value` for a press button event is 1 and for a release button event is 0.

Though this:

```
if (js_event.type == JS_EVENT_BUTTON) {
        buttons_state ^= (1 << js_event.number);
}
```

may work well if you handle JS_EVENT_INIT events separately,

```
if ((js_event.type & ~JS_EVENT_INIT) == JS_EVENT_BUTTON) {
        if (js_event.value)
                buttons_state |= (1 << js_event.number);
        else
                buttons_state &= ~(1 << js_event.number);
}
```

is much safer since it can't lose sync with the driver. As you would have to write a separate handler for JS_EVENT_INIT events in the first snippet, this ends up being shorter.

js_event.time

The time an event was generated is stored in `js_event.time`. It's a time in milliseconds since ... well, since sometime in the past. This eases the task of detecting double clicks, figuring out if movement of axis and button presses happened at the same time, and similar.

Reading

If you open the device in blocking mode, a read will block (that is, wait) forever until an event is generated and effectively read. There are two alternatives if you can't afford to wait forever (which is, admittedly, a long time;)

1. use select to wait until there's data to be read on fd, or until it timeouts. There's a good example on the select(2) man page.
2. open the device in non-blocking mode (O_NONBLOCK)

O_NONBLOCK

If read returns -1 when reading in O_NONBLOCK mode, this isn't necessarily a "real" error (check errno(3)); it can just mean there are no events pending to be read on the driver queue. You should read all events on the queue (that is, until you get a -1).

For example,

```
while (1) {
        while (read (fd, &e, sizeof(e)) > 0) {
                process_event (e);
        }
        /* EAGAIN is returned when the queue is empty */
        if (errno != EAGAIN) {
                /* error */
        }
        /* do something interesting with processed events */
}
```

One reason for emptying the queue is that if it gets full you'll start missing events since the queue is finite, and older events will get overwritten.

The other reason is that you want to know all what happened, and not delay the processing till later.

Why can get the queue full? Because you don't empty the queue as mentioned, or because too much time elapses from one read to another and too many events to store in the queue get generated. Note that high system load may contribute to space those reads even more.

If time between reads is enough to fill the queue and lose an event, the driver will switch to startup mode and next time you read it, synthetic events (JS_EVENT_INIT) will be generated to inform you of the actual state of the joystick.

Note:

As of version 1.2.8, the queue is circular and able to hold 64 events. You can increment this size bumping up JS_BUFF_SIZE in joystick.h and recompiling the driver.

In the above code, you might as well want to read more than one event at a time using the typical read(2) functionality. For that, you would replace the read above with something like:

```
struct js_event mybuffer[0xff];
int i = read (fd, mybuffer, sizeof(mybuffer));
```

In this case, read would return -1 if the queue was empty, or some other value in which the number of events read would be i / sizeof(js_event) Again, if the buffer was full, it's a good idea to process the events and keep reading it until you empty the driver queue.

IOCTLs

The joystick driver defines the following ioctl(2) operations:

```
                        /* function                    3rd arg  */
#define JSIOCGAXES      /* get number of axes          char     */
#define JSIOCGBUTTONS   /* get number of buttons       char     */
#define JSIOCGVERSION   /* get driver version          int      */
#define JSIOCGNAME(len) /* get identifier string       char     */
#define JSIOCSCORR      /* set correction values       &js_corr */
#define JSIOCGCORR      /* get correction values       &js_corr */
```

For example, to read the number of axes:

```
char number_of_axes;
ioctl (fd, JSIOCGAXES, &number_of_axes);
```

JSIOGCVERSION

JSIOGCVERSION is a good way to check in run-time whether the running driver is 1.0+ and supports the event interface. If it is not, the IOCTL will fail. For a compile-time decision, you can test the JS_VERSION symbol:

```
#ifdef JS_VERSION
#if JS_VERSION > 0xsomething
```

JSIOCGNAME

JSIOCGNAME(len) allows you to get the name string of the joystick - the same as is being printed at boot time. The 'len' argument is the length of the buffer provided by the application asking for the name. It is used to avoid possible overrun should the name be too long:

```
char name[128];
if (ioctl(fd, JSIOCGNAME(sizeof(name)), name) < 0)
        strncpy(name, "Unknown", sizeof(name));
printf("Name: %s\n", name);
```

JSIOC[SG]CORR

For usage on JSIOC[SG]CORR I suggest you to look into jscal.c They are not needed in a normal program, only in joystick calibration software such as jscal or kcmjoy. These IOCTLs and data types aren't considered to be in the stable part of the API, and therefore may change without warning in following releases of the driver.

Both JSIOCSCORR and JSIOCGCORR expect &js_corr to be able to hold information for all axis. That is, struct js_corr corr[MAX_AXIS];

struct js_corr is defined as:

```
struct js_corr {
        __s32 coef[8];
        __u16 prec;
        __u16 type;
};
```

and type:

```
#define JS_CORR_NONE            0x00    /* returns raw values */
#define JS_CORR_BROKEN          0x01    /* broken line */
```

Backward compatibility

The 0.x joystick driver API is quite limited and its usage is deprecated. The driver offers backward compatibility, though. Here's a quick summary:

```
struct JS_DATA_TYPE js;
while (1) {
        if (read (fd, &js, JS_RETURN) != JS_RETURN) {
                /* error */
        }
        usleep (1000);
}
```

As you can figure out from the example, the read returns immediately, with the actual state of the joystick:

```
struct JS_DATA_TYPE {
        int buttons;    /* immediate button state */
        int x;          /* immediate x axis value */
        int y;          /* immediate y axis value */
};
```

and JS_RETURN is defined as:

```
#define JS_RETURN       sizeof(struct JS_DATA_TYPE)
```

To test the state of the buttons,

```
first_button_state  = js.buttons & 1;
second_button_state = js.buttons & 2;
```

The axis values do not have a defined range in the original 0.x driver, except for that the values are non-negative. The 1.2.8+ drivers use a fixed range for reporting the values, 1 being the minimum, 128 the center, and 255 maximum value.

The v0.8.0.2 driver also had an interface for 'digital joysticks', (now called Multisystem joysticks in this driver), under /dev/djsX. This driver doesn't try to be compatible with that interface.

Final Notes

```
____/|        Comments, additions, and specially corrections are welcome.
\ o.O|        Documentation valid for at least version 1.2.8 of the joystick
 =(_)=        driver and as usual, the ultimate source for documentation is
   U          to "Use The Source Luke" or, at your convenience, Vojtech ;)
```

uinput module

Introduction

uinput is a kernel module that makes it possible to emulate input devices from userspace. By writing to /dev/uinput (or /dev/input/uinput) device, a process can create a virtual input device with specific capabilities. Once this virtual device is created, the process can send events through it, that will be delivered to userspace and in-kernel consumers.

Interface

```
linux/uinput.h
```

The uinput header defines ioctls to create, set up, and destroy virtual devices.

libevdev

libevdev is a wrapper library for evdev devices that provides interfaces to create uinput devices and send events. libevdev is less error-prone than accessing uinput directly, and should be considered for new software.

For examples and more information about libevdev: https://www.freedesktop.org/software/libevdev/doc/latest/

Examples

Keyboard events

This first example shows how to create a new virtual device, and how to send a key event. All default imports and error handlers were removed for the sake of simplicity.

```
#include <linux/uinput.h>

void emit(int fd, int type, int code, int val)
{
   struct input_event ie;

   ie.type = type;
   ie.code = code;
   ie.value = val;
   /* timestamp values below are ignored */
   ie.time.tv_sec = 0;
   ie.time.tv_usec = 0;

   write(fd, &ie, sizeof(ie));
}

int main(void)
{
   struct uinput_setup usetup;

   int fd = open("/dev/uinput", O_WRONLY | O_NONBLOCK);


   /*
    * The ioctls below will enable the device that is about to be
    * created, to pass key events, in this case the space key.
    */
   ioctl(fd, UI_SET_EVBIT, EV_KEY);
   ioctl(fd, UI_SET_KEYBIT, KEY_SPACE);

   memset(&usetup, 0, sizeof(usetup));
   usetup.id.bustype = BUS_USB;
   usetup.id.vendor = 0x1234; /* sample vendor */
   usetup.id.product = 0x5678; /* sample product */
   strcpy(usetup.name, "Example device");

   ioctl(fd, UI_DEV_SETUP, &usetup);
   ioctl(fd, UI_DEV_CREATE);
```

```
   /*
    * On UI_DEV_CREATE the kernel will create the device node for this
    * device. We are inserting a pause here so that userspace has time
    * to detect, initialize the new device, and can start listening to
    * the event, otherwise it will not notice the event we are about
    * to send. This pause is only needed in our example code!
    */
   sleep(1);

   /* Key press, report the event, send key release, and report again */
   emit(fd, EV_KEY, KEY_SPACE, 1);
   emit(fd, EV_SYN, SYN_REPORT, 0);
   emit(fd, EV_KEY, KEY_SPACE, 0);
   emit(fd, EV_SYN, SYN_REPORT, 0);

   /*
    * Give userspace some time to read the events before we destroy the
    * device with UI_DEV_DESTOY.
    */
   sleep(1);

   ioctl(fd, UI_DEV_DESTROY);
   close(fd);

   return 0;
}
```

Mouse movements

This example shows how to create a virtual device that behaves like a physical mouse.

```
#include <linux/uinput.h>

/* emit function is identical to of the first example */

int main(void)
{
   struct uinput_setup usetup;
   int i = 50;

   int fd = open("/dev/uinput", O_WRONLY | O_NONBLOCK);

   /* enable mouse button left and relative events */
   ioctl(fd, UI_SET_EVBIT, EV_KEY);
   ioctl(fd, UI_SET_KEYBIT, BTN_LEFT);

   ioctl(fd, UI_SET_EVBIT, EV_REL);
   ioctl(fd, UI_SET_RELBIT, REL_X);
   ioctl(fd, UI_SET_RELBIT, REL_Y);

   memset(&usetup, 0, sizeof(usetup));
   usetup.id.bustype = BUS_USB;
   usetup.id.vendor = 0x1234; /* sample vendor */
   usetup.id.product = 0x5678; /* sample product */
   strcpy(usetup.name, "Example device");

   ioctl(fd, UI_DEV_SETUP, &usetup);
   ioctl(fd, UI_DEV_CREATE);

   /*
```

```
     * On UI_DEV_CREATE the kernel will create the device node for this
     * device. We are inserting a pause here so that userspace has time
     * to detect, initialize the new device, and can start listening to
     * the event, otherwise it will not notice the event we are about
     * to send. This pause is only needed in our example code!
     */
    sleep(1);

    /* Move the mouse diagonally, 5 units per axis */
    while (i--) {
        emit(fd, EV_REL, REL_X, 5);
        emit(fd, EV_REL, REL_Y, 5);
        emit(fd, EV_SYN, SYN_REPORT, 0);
        usleep(15000);
    }

    /*
     * Give userspace some time to read the events before we destroy the
     * device with UI_DEV_DESTOY.
     */
    sleep(1);

    ioctl(fd, UI_DEV_DESTROY);
    close(fd);

    return 0;
}
```

uinput old interface

Before uinput version 5, there wasn't a dedicated ioctl to set up a virtual device. Programs supportinf older versions of uinput interface need to fill a uinput_user_dev structure and write it to the uinput file descriptor to configure the new uinput device. New code should not use the old interface but interact with uinput via ioctl calls, or use libevdev.

```
#include <linux/uinput.h>

/* emit function is identical to of the first example */

int main(void)
{
    struct uinput_user_dev uud;
    int version, rc, fd;

    fd = open("/dev/uinput", O_WRONLY | O_NONBLOCK);
    rc = ioctl(fd, UI_GET_VERSION, &version);

    if (rc == 0 && version >= 5) {
        /* use UI_DEV_SETUP */
        return 0;
    }

    /*
     * The ioctls below will enable the device that is about to be
     * created, to pass key events, in this case the space key.
     */
    ioctl(fd, UI_SET_EVBIT, EV_KEY);
    ioctl(fd, UI_SET_KEYBIT, KEY_SPACE);

    memset(&uud, 0, sizeof(uud));
    snprintf(uud.name, UINPUT_MAX_NAME_SIZE, "uinput old interface");
```

```
    write(fd, &uud, sizeof(uud));

    ioctl(fd, UI_DEV_CREATE);

    /*
     * On UI_DEV_CREATE the kernel will create the device node for this
     * device. We are inserting a pause here so that userspace has time
     * to detect, initialize the new device, and can start listening to
     * the event, otherwise it will not notice the event we are about
     * to send. This pause is only needed in our example code!
     */
    sleep(1);

    /* Key press, report the event, send key release, and report again */
    emit(fd, EV_KEY, KEY_SPACE, 1);
    emit(fd, EV_SYN, SYN_REPORT, 0);
    emit(fd, EV_KEY, KEY_SPACE, 0);
    emit(fd, EV_SYN, SYN_REPORT, 0);

    /*
     * Give userspace some time to read the events before we destroy the
     * device with UI_DEV_DESTOY.
     */
    sleep(1);

    ioctl(fd, UI_DEV_DESTROY);

    close(fd);
    return 0;
}
```

The userio Protocol

Sponsored by Red Hat

Introduction

This module is intended to try to make the lives of input driver developers easier by allowing them to test various serio devices (mainly the various touchpads found on laptops) without having to have the physical device in front of them. userio accomplishes this by allowing any privileged userspace program to directly interact with the kernel's serio driver and control a virtual serio port from there.

Usage overview

In order to interact with the userio kernel module, one simply opens the /dev/userio character device in their applications. Commands are sent to the kernel module by writing to the device, and any data received from the serio driver is read as-is from the /dev/userio device. All of the structures and macros you need to interact with the device are defined in <linux/userio.h> and <linux/serio.h>.

Command Structure

The struct used for sending commands to /dev/userio is as follows:

```
struct userio_cmd {
        __u8 type;
        __u8 data;
};
```

type describes the type of command that is being sent. This can be any one of the USERIO_CMD macros defined in <linux/userio.h>. data is the argument that goes along with the command. In the event that the command doesn't have an argument, this field can be left untouched and will be ignored by the kernel. Each command should be sent by writing the struct directly to the character device. In the event that the command you send is invalid, an error will be returned by the character device and a more descriptive error will be printed to the kernel log. Only one command can be sent at a time, any additional data written to the character device after the initial command will be ignored.

To close the virtual serio port, just close /dev/userio.

Commands

USERIO_CMD_REGISTER

Registers the port with the serio driver and begins transmitting data back and forth. Registration can only be performed once a port type is set with USERIO_CMD_SET_PORT_TYPE. Has no argument.

USERIO_CMD_SET_PORT_TYPE

Sets the type of port we're emulating, where data is the port type being set. Can be any of the macros from <linux/serio.h>. For example: SERIO_8042 would set the port type to be a normal PS/2 port.

USERIO_CMD_SEND_INTERRUPT

Sends an interrupt through the virtual serio port to the serio driver, where data is the interrupt data being sent.

Userspace tools

The userio userspace tools are able to record PS/2 devices using some of the debugging information from i8042, and play back the devices on /dev/userio. The latest version of these tools can be found at:

https://github.com/Lyude/ps2emu

CHAPTER

TWO

LINUX INPUT SUBSYSTEM KERNEL API

Table of Contents

Creating an input device driver

The simplest example

Here comes a very simple example of an input device driver. The device has just one button and the button is accessible at i/o port BUTTON_PORT. When pressed or released a BUTTON_IRQ happens. The driver could look like:

```
#include <linux/input.h>
#include <linux/module.h>
#include <linux/init.h>

#include <asm/irq.h>
#include <asm/io.h>

static struct input_dev *button_dev;

static irqreturn_t button_interrupt(int irq, void *dummy)
{
        input_report_key(button_dev, BTN_0, inb(BUTTON_PORT) & 1);
        input_sync(button_dev);
        return IRQ_HANDLED;
}

static int __init button_init(void)
{
        int error;

        if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
                printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
                return -EBUSY;
        }

        button_dev = input_allocate_device();
        if (!button_dev) {
                printk(KERN_ERR "button.c: Not enough memory\n");
                error = -ENOMEM;
                goto err_free_irq;
        }

        button_dev->evbit[0] = BIT_MASK(EV_KEY);
        button_dev->keybit[BIT_WORD(BTN_0)] = BIT_MASK(BTN_0);

        error = input_register_device(button_dev);
```

```
        if (error) {
                printk(KERN_ERR "button.c: Failed to register device\n");
                goto err_free_dev;
        }

        return 0;

err_free_dev:
        input_free_device(button_dev);
err_free_irq:
        free_irq(BUTTON_IRQ, button_interrupt);
        return error;
}

static void __exit button_exit(void)
{
        input_unregister_device(button_dev);
        free_irq(BUTTON_IRQ, button_interrupt);
}

module_init(button_init);
module_exit(button_exit);
```

What the example does

First it has to include the <linux/input.h> file, which interfaces to the input subsystem. This provides all the definitions needed.

In the _init function, which is called either upon module load or when booting the kernel, it grabs the required resources (it should also check for the presence of the device).

Then it allocates a new input device structure with input_allocate_device() and sets up input bitfields. This way the device driver tells the other parts of the input systems what it is - what events can be generated or accepted by this input device. Our example device can only generate EV_KEY type events, and from those only BTN_0 event code. Thus we only set these two bits. We could have used:

```
set_bit(EV_KEY, button_dev.evbit);
set_bit(BTN_0, button_dev.keybit);
```

as well, but with more than single bits the first approach tends to be shorter.

Then the example driver registers the input device structure by calling:

```
input_register_device(&button_dev);
```

This adds the button_dev structure to linked lists of the input driver and calls device handler modules _connect functions to tell them a new input device has appeared. input_register_device() may sleep and therefore must not be called from an interrupt or with a spinlock held.

While in use, the only used function of the driver is:

```
button_interrupt()
```

which upon every interrupt from the button checks its state and reports it via the:

```
input_report_key()
```

call to the input system. There is no need to check whether the interrupt routine isn't reporting two same value events (press, press for example) to the input system, because the input_report_* functions check that themselves.

Then there is the:

```
input_sync()
```

call to tell those who receive the events that we've sent a complete report. This doesn't seem important in the one button case, but is quite important for for example mouse movement, where you don't want the X and Y values to be interpreted separately, because that'd result in a different movement.

dev->open() and dev->close()

In case the driver has to repeatedly poll the device, because it doesn't have an interrupt coming from it and the polling is too expensive to be done all the time, or if the device uses a valuable resource (eg. interrupt), it can use the open and close callback to know when it can stop polling or release the interrupt and when it must resume polling or grab the interrupt again. To do that, we would add this to our example driver:

```
static int button_open(struct input_dev *dev)
{
        if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
                printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
                return -EBUSY;
        }

        return 0;
}

static void button_close(struct input_dev *dev)
{
        free_irq(IRQ_AMIGA_VERTB, button_interrupt);
}

static int __init button_init(void)
{
        ...
        button_dev->open = button_open;
        button_dev->close = button_close;
        ...
}
```

Note that input core keeps track of number of users for the device and makes sure that dev->open() is called only when the first user connects to the device and that dev->close() is called when the very last user disconnects. Calls to both callbacks are serialized.

The open() callback should return a 0 in case of success or any nonzero value in case of failure. The close() callback (which is void) must always succeed.

Basic event types

The most simple event type is EV_KEY, which is used for keys and buttons. It's reported to the input system via:

```
input_report_key(struct input_dev *dev, int code, int value)
```

See uapi/linux/input-event-codes.h for the allowable values of code (from 0 to KEY_MAX). Value is interpreted as a truth value, ie any nonzero value means key pressed, zero value means key released. The input code generates events only in case the value is different from before.

In addition to EV_KEY, there are two more basic event types: EV_REL and EV_ABS. They are used for relative and absolute values supplied by the device. A relative value may be for example a mouse movement in the X axis. The mouse reports it as a relative difference from the last position, because it doesn't

have any absolute coordinate system to work in. Absolute events are namely for joysticks and digitizers - devices that do work in an absolute coordinate systems.

Having the device report EV_REL buttons is as simple as with EV_KEY, simply set the corresponding bits and call the:

```
input_report_rel(struct input_dev *dev, int code, int value)
```

function. Events are generated only for nonzero value.

However EV_ABS requires a little special care. Before calling input_register_device, you have to fill additional fields in the input_dev struct for each absolute axis your device has. If our button device had also the ABS_X axis:

```
button_dev.absmin[ABS_X] = 0;
button_dev.absmax[ABS_X] = 255;
button_dev.absfuzz[ABS_X] = 4;
button_dev.absflat[ABS_X] = 8;
```

Or, you can just say:

```
input_set_abs_params(button_dev, ABS_X, 0, 255, 4, 8);
```

This setting would be appropriate for a joystick X axis, with the minimum of 0, maximum of 255 (which the joystick *must* be able to reach, no problem if it sometimes reports more, but it must be able to always reach the min and max values), with noise in the data up to +- 4, and with a center flat position of size 8.

If you don't need absfuzz and absflat, you can set them to zero, which mean that the thing is precise and always returns to exactly the center position (if it has any).

BITS_TO_LONGS(), BIT_WORD(), BIT_MASK()

These three macros from bitops.h help some bitfield computations:

```
BITS_TO_LONGS(x) - returns the length of a bitfield array in longs for
                   x bits
BIT_WORD(x)      - returns the index in the array in longs for bit x
BIT_MASK(x)      - returns the index in a long for bit x
```

The id* and name fields

The dev->name should be set before registering the input device by the input device driver. It's a string like 'Generic button device' containing a user friendly name of the device.

The id* fields contain the bus ID (PCI, USB, ...), vendor ID and device ID of the device. The bus IDs are defined in input.h. The vendor and device ids are defined in pci_ids.h, usb_ids.h and similar include files. These fields should be set by the input device driver before registering it.

The idtype field can be used for specific information for the input device driver.

The id and name fields can be passed to userland via the evdev interface.

The keycode, keycodemax, keycodesize fields

These three fields should be used by input devices that have dense keymaps. The keycode is an array used to map from scancodes to input system keycodes. The keycode max should contain the size of the array and keycodesize the size of each entry in it (in bytes).

Userspace can query and alter current scancode to keycode mappings using EVIOCGKEYCODE and EVIOCSKEYCODE ioctls on corresponding evdev interface. When a device has all 3 aforementioned fields filled in, the driver may rely on kernel's default implementation of setting and querying keycode mappings.

dev->getkeycode() and dev->setkeycode()

getkeycode() and setkeycode() callbacks allow drivers to override default keycode/keycodesize/keycodemax mapping mechanism provided by input core and implement sparse keycode maps.

Key autorepeat

... is simple. It is handled by the input.c module. Hardware autorepeat is not used, because it's not present in many devices and even where it is present, it is broken sometimes (at keyboards: Toshiba notebooks). To enable autorepeat for your device, just set EV_REP in dev->evbit. All will be handled by the input system.

Other event types, handling output events

The other event types up to now are:

- EV_LED - used for the keyboard LEDs.
- EV_SND - used for keyboard beeps.

They are very similar to for example key events, but they go in the other direction - from the system to the input device driver. If your input device driver can handle these events, it has to set the respective bits in evbit, *and* also the callback routine:

```
button_dev->event = button_event;

int button_event(struct input_dev *dev, unsigned int type,
                 unsigned int code, int value)
{
        if (type == EV_SND && code == SND_BELL) {
                outb(value, BUTTON_BELL);
                return 0;
        }
        return -1;
}
```

This callback routine can be called from an interrupt or a BH (although that isn't a rule), and thus must not sleep, and must not take too long to finish.

Programming gameport drivers

A basic classic gameport

If the gameport doesn't provide more than the inb()/outb() functionality, the code needed to register it with the joystick drivers is simple:

```
struct gameport gameport;

gameport.io = MY_IO_ADDRESS;
gameport_register_port(&gameport);
```

Make sure struct gameport is initialized to 0 in all other fields. The gameport generic code will take care of the rest.

If your hardware supports more than one io address, and your driver can choose which one to program the hardware to, starting from the more exotic addresses is preferred, because the likelihood of clashing with the standard 0x201 address is smaller.

Eg. if your driver supports addresses 0x200, 0x208, 0x210 and 0x218, then 0x218 would be the address of first choice.

If your hardware supports a gameport address that is not mapped to ISA io space (is above 0x1000), use that one, and don't map the ISA mirror.

Also, always request_region() on the whole io space occupied by the gameport. Although only one ioport is really used, the gameport usually occupies from one to sixteen addresses in the io space.

Please also consider enabling the gameport on the card in the ->open() callback if the io is mapped to ISA space - this way it'll occupy the io space only when something really is using it. Disable it again in the ->close() callback. You also can select the io address in the ->open() callback, so that it doesn't fail if some of the possible addresses are already occupied by other gameports.

Memory mapped gameport

When a gameport can be accessed through MMIO, this way is preferred, because it is faster, allowing more reads per second. Registering such a gameport isn't as easy as a basic IO one, but not so much complex:

```
struct gameport gameport;

void my_trigger(struct gameport *gameport)
{
        my_mmio = 0xff;
}

unsigned char my_read(struct gameport *gameport)
{
        return my_mmio;
}

gameport.read = my_read;
gameport.trigger = my_trigger;
gameport_register_port(&gameport);
```

Cooked mode gameport

There are gameports that can report the axis values as numbers, that means the driver doesn't have to measure them the old way - an ADC is built into the gameport. To register a cooked gameport:

```
struct gameport gameport;

int my_cooked_read(struct gameport *gameport, int *axes, int *buttons)
{
        int i;

        for (i = 0; i < 4; i++)
                axes[i] = my_mmio[i];
        buttons[i] = my_mmio[4];
}

int my_open(struct gameport *gameport, int mode)
{
        return -(mode != GAMEPORT_MODE_COOKED);
}

gameport.cooked_read = my_cooked_read;
gameport.open = my_open;
gameport.fuzz = 8;
gameport_register_port(&gameport);
```

The only confusing thing here is the fuzz value. Best determined by experimentation, it is the amount of noise in the ADC data. Perfect gameports can set this to zero, most common have fuzz between 8 and 32. See analog.c and input.c for handling of fuzz - the fuzz value determines the size of a gaussian filter window that is used to eliminate the noise in the data.

More complex gameports

Gameports can support both raw and cooked modes. In that case combine either examples 1+2 or 1+3. Gameports can support internal calibration - see below, and also lightning.c and analog.c on how that works. If your driver supports more than one gameport instance simultaneously, use the ->private member of the gameport struct to point to your data.

Unregistering a gameport

Simple:

```
gameport_unregister_port(&gameport);
```

The gameport structure

Note:

This section is outdated. There are several fields here that don't match what's there at include/linux/gameport.h.

```
struct gameport {

    void *private;
```

A private pointer for free use in the gameport driver. (Not the joystick driver!)

```
int number;
```

Number assigned to the gameport when registered. Informational purpose only.

```
int io;
```

I/O address for use with raw mode. You have to either set this, or ->read() to some value if your gameport supports raw mode.

```
int speed;
```

Raw mode speed of the gameport reads in thousands of reads per second.

```
int fuzz;
```

If the gameport supports cooked mode, this should be set to a value that represents the amount of noise in the data. See *Cooked mode gameport* .

```
void (*trigger)(struct gameport *);
```

Trigger. This function should trigger the ns558 oneshots. If set to NULL, outb(0xff, io) will be used.

```
unsigned char (*read)(struct gameport *);
```

Read the buttons and ns558 oneshot bits. If set to NULL, inb(io) will be used instead.

```
int (*cooked_read)(struct gameport *, int *axes, int *buttons);
```

If the gameport supports cooked mode, it should point this to its cooked read function. It should fill axes[0..3] with four values of the joystick axes and buttons[0] with four bits representing the buttons.

```
int (*calibrate)(struct gameport *, int *axes, int *max);
```

Function for calibrating the ADC hardware. When called, axes[0..3] should be pre-filled by cooked data by the caller, max[0..3] should be pre-filled with expected maximums for each axis. The calibrate() function should set the sensitivity of the ADC hardware so that the maximums fit in its range and recompute the axes[] values to match the new sensitivity or re-read them from the hardware so that they give valid values.

```
int (*open)(struct gameport *, int mode);
```

Open() serves two purposes. First a driver either opens the port in raw or in cooked mode, the open() callback can decide which modes are supported. Second, resource allocation can happen here. The port can also be enabled here. Prior to this call, other fields of the gameport struct (namely the io member) need not to be valid.

```
void (*close)(struct gameport *);
```

Close() should free the resources allocated by open, possibly disabling the gameport.

```
struct gameport_dev *dev;
struct gameport *next;
```

For internal use by the gameport layer.

```
};
```

Enjoy!

Keyboard notifier

One can use register_keyboard_notifier to get called back on keyboard events (see kbd_keycode() function for details). The passed structure is keyboard_notifier_param:

- ‘vc’ always provide the VC for which the keyboard event applies;
- ‘down’ is 1 for a key press event, 0 for a key release;
- ‘shift’ is the current modifier state, mask bit indexes are KG_*;
- ‘value’ depends on the type of event.
- KBD_KEYCODE events are always sent before other events, value is the keycode.
- KBD_UNBOUND_KEYCODE events are sent if the keycode is not bound to a keysym. value is the keycode.
- KBD_UNICODE events are sent if the keycode -> keysym translation produced a unicode character. value is the unicode value.
- KBD_KEYSYM events are sent if the keycode -> keysym translation produced a non-unicode character. value is the keysym.
- KBD_POST_KEYSYM events are sent after the treatment of non-unicode keysyms. That permits one to inspect the resulting LEDs for instance.

For each kind of event but the last, the callback may return NOTIFY_STOP in order to “eat” the event: the notify loop is stopped and the keyboard event is dropped.

In a rough C snippet, we have:

```
kbd_keycode(keycode) {
    ...
    params.value = keycode;
    if (notifier_call_chain(KBD_KEYCODE,&params) == NOTIFY_STOP)
        || !bound) {
            notifier_call_chain(KBD_UNBOUND_KEYCODE,&params);
            return;
    }

    if (unicode) {
            param.value = unicode;
            if (notifier_call_chain(KBD_UNICODE,&params) == NOTIFY_STOP)
                    return;
            emit unicode;
            return;
    }

    params.value = keysym;
    if (notifier_call_chain(KBD_KEYSYM,&params) == NOTIFY_STOP)
            return;
    apply keysym;
    notifier_call_chain(KBD_POST_KEYSYM,&params);
}
```

Note:

This notifier is usually called from interrupt context.

CHAPTER

THREE

DRIVER-SPECIFIC DOCUMENTATION

This section provides information about various devices supported by the Linux kernel, their protocols, and driver details.

ALPS Touchpad Protocol

Introduction

Currently the ALPS touchpad driver supports seven protocol versions in use by ALPS touchpads, called versions 1, 2, 3, 4, 5, 6, 7 and 8.

Since roughly mid-2010 several new ALPS touchpads have been released and integrated into a variety of laptops and netbooks. These new touchpads have enough behavior differences that the alps_model_data definition table, describing the properties of the different versions, is no longer adequate. The design choices were to re-define the alps_model_data table, with the risk of regression testing existing devices, or isolate the new devices outside of the alps_model_data table. The latter design choice was made. The new touchpad signatures are named: “Rushmore”, “Pinnacle”, and “Dolphin”, which you will see in the alps.c code. For the purposes of this document, this group of ALPS touchpads will generically be called “new ALPS touchpads”.

We experimented with probing the ACPI interface _HID (Hardware ID)/_CID (Compatibility ID) definition as a way to uniquely identify the different ALPS variants but there did not appear to be a 1:1 mapping. In fact, it appeared to be an m:n mapping between the _HID and actual hardware type.

Detection

All ALPS touchpads should respond to the “E6 report” command sequence: E8-E6-E6-E6-E9. An ALPS touchpad should respond with either 00-00-0A or 00-00-64 if no buttons are pressed. The bits 0-2 of the first byte will be 1s if some buttons are pressed.

If the E6 report is successful, the touchpad model is identified using the “E7 report” sequence: E8-E7-E7-E7-E9. The response is the model signature and is matched against known models in the alps_model_data_array.

For older touchpads supporting protocol versions 3 and 4, the E7 report model signature is always 73-02-64. To differentiate between these versions, the response from the “Enter Command Mode” sequence must be inspected as described below.

The new ALPS touchpads have an E7 signature of 73-03-50 or 73-03-0A but seem to be better differentiated by the EC Command Mode response.

Command Mode

Protocol versions 3 and 4 have a command mode that is used to read and write one-byte device registers in a 16-bit address space. The command sequence EC-EC-EC-E9 places the device in command mode,

and the device will respond with 88-07 followed by a third byte. This third byte can be used to determine whether the devices uses the version 3 or 4 protocol.

To exit command mode, PSMOUSE_CMD_SETSTREAM (EA) is sent to the touchpad.

While in command mode, register addresses can be set by first sending a specific command, either EC for v3 devices or F5 for v4 devices. Then the address is sent one nibble at a time, where each nibble is encoded as a command with optional data. This encoding differs slightly between the v3 and v4 protocols.

Once an address has been set, the addressed register can be read by sending PSMOUSE_CMD_GETINFO (E9). The first two bytes of the response contains the address of the register being read, and the third contains the value of the register. Registers are written by writing the value one nibble at a time using the same encoding used for addresses.

For the new ALPS touchpads, the EC command is used to enter command mode. The response in the new ALPS touchpads is significantly different, and more important in determining the behavior. This code has been separated from the original alps_model_data table and put in the alps_identify function. For example, there seem to be two hardware init sequences for the “Dolphin” touchpads as determined by the second byte of the EC response.

Packet Format

In the following tables, the following notation is used:

```
CAPITALS = stick, miniscules = touchpad
```

?’s can have different meanings on different models, such as wheel rotation, extra buttons, stick buttons on a dualpoint, etc.

PS/2 packet format

```
byte 0:  0    0 YSGN XSGN    1    M    R    L
byte 1: X7   X6   X5   X4   X3   X2   X1   X0
byte 2: Y7   Y6   Y5   Y4   Y3   Y2   Y1   Y0
```

Note that the device never signals overflow condition.

For protocol version 2 devices when the trackpoint is used, and no fingers are on the touchpad, the M R L bits signal the combined status of both the pointingstick and touchpad buttons.

ALPS Absolute Mode - Protocol Version 1

```
byte 0:  1    0    0    0    1   x9   x8   x7
byte 1:  0   x6   x5   x4   x3   x2   x1   x0
byte 2:  0    ?    ?    l    r    ?  fin  ges
byte 3:  0    ?    ?    ?    ?   y9   y8   y7
byte 4:  0   y6   y5   y4   y3   y2   y1   y0
byte 5:  0   z6   z5   z4   z3   z2   z1   z0
```

ALPS Absolute Mode - Protocol Version 2

```
byte 0:  1    ?    ?    ?    1  PSM  PSR  PSL
byte 1:  0   x6   x5   x4   x3   x2   x1   x0
byte 2:  0  x10   x9   x8   x7    ?  fin  ges
byte 3:  0   y9   y8   y7    1    M    R    L
byte 4:  0   y6   y5   y4   y3   y2   y1   y0
byte 5:  0   z6   z5   z4   z3   z2   z1   z0
```

Protocol Version 2 DualPoint devices send standard PS/2 mouse packets for the DualPoint Stick. The M, R and L bits signal the combined status of both the pointingstick and touchpad buttons, except for Dell dualpoint devices where the pointingstick buttons get reported separately in the PSM, PSR and PSL bits.

Dualpoint device – interleaved packet format

```
byte 0:    1    1    0    0    1    1    1    1
byte 1:    0   x6   x5   x4   x3   x2   x1   x0
byte 2:    0  x10   x9   x8   x7    0  fin  ges
byte 3:    0    0 YSGN XSGN    1    1    1    1
byte 4:   X7   X6   X5   X4   X3   X2   X1   X0
byte 5:   Y7   Y6   Y5   Y4   Y3   Y2   Y1   Y0
byte 6:    0   y9   y8   y7    1    m    r    l
byte 7:    0   y6   y5   y4   y3   y2   y1   y0
byte 8:    0   z6   z5   z4   z3   z2   z1   z0
```

Devices which use the interleaving format normally send standard PS/2 mouse packets for the DualPoint Stick + ALPS Absolute Mode packets for the touchpad, switching to the interleaved packet format when both the stick and the touchpad are used at the same time.

ALPS Absolute Mode - Protocol Version 3

ALPS protocol version 3 has three different packet formats. The first two are associated with touchpad events, and the third is associated with trackstick events.

The first type is the touchpad position packet:

```
byte 0:    1    ?   x1   x0    1    1    1    1
byte 1:    0  x10   x9   x8   x7   x6   x5   x4
byte 2:    0  y10   y9   y8   y7   y6   y5   y4
byte 3:    0    M    R    L    1    m    r    l
byte 4:    0   mt   x3   x2   y3   y2   y1   y0
byte 5:    0   z6   z5   z4   z3   z2   z1   z0
```

Note that for some devices the trackstick buttons are reported in this packet, and on others it is reported in the trackstick packets.

The second packet type contains bitmaps representing the x and y axes. In the bitmaps a given bit is set if there is a finger covering that position on the given axis. Thus the bitmap packet can be used for low-resolution multi-touch data, although finger tracking is not possible. This packet also encodes the number of contacts (f1 and f0 in the table below):

```
byte 0:    1    1   x1   x0    1    1    1    1
byte 1:    0   x8   x7   x6   x5   x4   x3   x2
byte 2:    0   y7   y6   y5   y4   y3   y2   y1
byte 3:    0  y10   y9   y8    1    1    1    1
byte 4:    0  x14  x13  x12  x11  x10   x9   y0
byte 5:    0    1    ?    ?    ?    ?   f1   f0
```

This packet only appears after a position packet with the mt bit set, and usually only appears when there are two or more contacts (although occasionally it's seen with only a single contact).

The final v3 packet type is the trackstick packet:

```
byte 0:    1    1   x7   y7    1    1    1    1
byte 1:    0   x6   x5   x4   x3   x2   x1   x0
byte 2:    0   y6   y5   y4   y3   y2   y1   y0
byte 3:    0    1    0    0    1    0    0    0
byte 4:    0   z4   z3   z2   z1   z0    ?    ?
byte 5:    0    0    1    1    1    1    1    1
```

ALPS Absolute Mode - Protocol Version 4

Protocol version 4 has an 8-byte packet format:

```
byte 0:    1    ?   x1   x0    1    1    1    1
byte 1:    0  x10   x9   x8   x7   x6   x5   x4
byte 2:    0  y10   y9   y8   y7   y6   y5   y4
byte 3:    0    1   x3   x2   y3   y2   y1   y0
byte 4:    0    ?    ?    ?    1    ?    r    l
byte 5:    0   z6   z5   z4   z3   z2   z1   z0
byte 6:    bitmap data (described below)
byte 7:    bitmap data (described below)
```

The last two bytes represent a partial bitmap packet, with 3 full packets required to construct a complete bitmap packet. Once assembled, the 6-byte bitmap packet has the following format:

```
byte 0:    0    1   x7   x6   x5   x4   x3   x2
byte 1:    0   x1   x0   y4   y3   y2   y1   y0
byte 2:    0    0    ?  x14  x13  x12  x11  x10
byte 3:    0   x9   x8   y9   y8   y7   y6   y5
byte 4:    0    0    0    0    0    0    0    0
byte 5:    0    0    0    0    0    0    0  y10
```

There are several things worth noting here.

1. In the bitmap data, bit 6 of byte 0 serves as a sync byte to identify the first fragment of a bitmap packet.
2. The bitmaps represent the same data as in the v3 bitmap packets, although the packet layout is different.
3. There doesn't seem to be a count of the contact points anywhere in the v4 protocol packets. Deriving a count of contact points must be done by analyzing the bitmaps.
4. There is a 3 to 1 ratio of position packets to bitmap packets. Therefore MT position can only be updated for every third ST position update, and the count of contact points can only be updated every third packet as well.

So far no v4 devices with tracksticks have been encountered.

ALPS Absolute Mode - Protocol Version 5

This is basically Protocol Version 3 but with different logic for packet decode. It uses the same alps_process_touchpad_packet_v3 call with a specialized decode_fields function pointer to correctly interpret the packets. This appears to only be used by the Dolphin devices.

For single-touch, the 6-byte packet format is:

```
byte 0:    1    1    0    0    1    0    0    0
byte 1:    0   x6   x5   x4   x3   x2   x1   x0
byte 2:    0   y6   y5   y4   y3   y2   y1   y0
byte 3:    0    M    R    L    1    m    r    l
byte 4:   y10  y9   y8   y7  x10   x9   x8   x7
byte 5:    0   z6   z5   z4   z3   z2   z1   z0
```

For mt, the format is:

```
byte 0:    1    1    1    n3   1   n2   n1   x24
byte 1:    1   y7   y6    y5  y4   y3   y2    y1
byte 2:    ?   x2   x1   y12 y11  y10   y9    y8
byte 3:    0  x23  x22   x21 x20  x19  x18   x17
byte 4:    0   x9   x8    x7  x6   x5   x4    x3
byte 5:    0  x16  x15   x14 x13  x12  x11   x10
```

ALPS Absolute Mode - Protocol Version 6

For trackstick packet, the format is:

```
byte 0:    1    1    1    1    1    1    1    1
byte 1:    0   X6   X5   X4   X3   X2   X1   X0
byte 2:    0   Y6   Y5   Y4   Y3   Y2   Y1   Y0
byte 3:    ?   Y7   X7    ?    ?    M    R    L
byte 4:   Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
byte 5:    0    1    1    1    1    1    1    1
```

For touchpad packet, the format is:

```
byte 0:    1    1    1    1    1    1    1    1
byte 1:    0    0    0    0   x3   x2   x1   x0
byte 2:    0    0    0    0   y3   y2   y1   y0
byte 3:    ?   x7   x6   x5   x4    ?    r    l
byte 4:    ?   y7   y6   y5   y4    ?    ?    ?
byte 5:   z7   z6   z5   z4   z3   z2   z1   z0
```

(v6 touchpad does not have middle button)

ALPS Absolute Mode - Protocol Version 7

For trackstick packet, the format is:

```
byte 0:    0    1    0    0    1    0    0    0
byte 1:    1    1    *    *    1    M    R    L
byte 2:   X7    1   X5   X4   X3   X2   X1   X0
byte 3:   Z6    1   Y6   X6    1   Y2   Y1   Y0
byte 4:   Y7    0   Y5   Y4   Y3    1    1    0
byte 5:  T&P    0   Z5   Z4   Z3   Z2   Z1   Z0
```

For touchpad packet, the format is:

```
        packet-fmt     b7     b6     b5     b4     b3     b2     b1     b0
byte 0: TWO & MULTI     L      1      R      M      1   Y0-2   Y0-1   Y0-0
byte 0: NEW             L      1   X1-5      1      1   Y0-2   Y0-1   Y0-0
byte 1:             Y0-10   Y0-9   Y0-8   Y0-7   Y0-6   Y0-5   Y0-4   Y0-3
byte 2:             X0-11      1  X0-10   X0-9   X0-8   X0-7   X0-6   X0-5
byte 3:             X1-11      1   X0-4   X0-3      1   X0-2   X0-1   X0-0
byte 4: TWO         X1-10    TWO   X1-9   X1-8   X1-7   X1-6   X1-5   X1-4
byte 4: MULTI       X1-10    TWO   X1-9   X1-8   X1-7   X1-6   Y1-5      1
byte 4: NEW         X1-10    TWO   X1-9   X1-8   X1-7   X1-6      0      0
byte 5: TWO & NEW   Y1-10      0   Y1-9   Y1-8   Y1-7   Y1-6   Y1-5   Y1-4
byte 5: MULTI       Y1-10      0   Y1-9   Y1-8   Y1-7   Y1-6    F-1    F-0

L:         Left button
R / M:     Non-clickpads: Right / Middle button
           Clickpads: When > 2 fingers are down, and some fingers
           are in the button area, then the 2 coordinates reported
           are for fingers outside the button area and these report
           extra fingers being present in the right / left button
           area. Note these fingers are not added to the F field!
           so if a TWO packet is received and R = 1 then there are
           3 fingers down, etc.
TWO:       1: Two touches present, byte 0/4/5 are in TWO fmt
           0: If byte 4 bit 0 is 1, then byte 0/4/5 are in MULTI fmt
              otherwise byte 0 bit 4 must be set and byte 0/4/5 are
              in NEW fmt
F:         Number of fingers - 3, 0 means 3 fingers, 1 means 4 ...
```

ALPS Absolute Mode - Protocol Version 8

Spoken by SS4 (73 03 14) and SS5 (73 03 28) hardware.

The packet type is given by the APD field, bits 4-5 of byte 3.

Touchpad packet (APD = 0x2):

```
          b7   b6   b5   b4   b3   b2   b1   b0
byte 0:  SWM  SWR  SWL    1    1    0    0   X7
byte 1:    0   X6   X5   X4   X3   X2   X1   X0
byte 2:    0   Y6   Y5   Y4   Y3   Y2   Y1   Y0
byte 3:    0  T&P    1    0    1    0    0   Y7
byte 4:    0   Z6   Z5   Z4   Z3   Z2   Z1   Z0
byte 5:    0    0    0    0    0    0    0    0
```

SWM, SWR, SWL: Middle, Right, and Left button states

Touchpad 1 Finger packet (APD = 0x0):

```
          b7   b6   b5   b4   b3   b2   b1   b0
byte 0:  SWM  SWR  SWL    1    1   X2   X1   X0
byte 1:   X9   X8   X7    1   X6   X5   X4   X3
byte 2:    0  X11  X10  LFB   Y3   Y2   Y1   Y0
byte 3:   Y5   Y4    0    0    1 TAPF2 TAPF1 TAPF0
byte 4:  Zv7  Y11  Y10    1   Y9   Y8   Y7   Y6
byte 5:  Zv6  Zv5  Zv4    0  Zv3  Zv2  Zv1  Zv0
```

TAPF: ??? LFB: ???

Touchpad 2 Finger packet (APD = 0x1):

```
          b7   b6   b5   b4   b3   b2   b1   b0
byte 0:  SWM  SWR  SWL    1    1  AX6  AX5  AX4
byte 1: AX11 AX10  AX9  AX8  AX7  AZ1  AY4  AZ0
byte 2: AY11 AY10  AY9  CONT AY8  AY7  AY6  AY5
byte 3:    0    0    0    1    1  BX6  BX5  BX4
byte 4: BX11 BX10  BX9  BX8  BX7  BZ1  BY4  BZ0
byte 5: BY11 BY10  BY9    0  BY8  BY7  BY5  BY5
```

CONT: A 3-or-4 Finger packet is to follow

Touchpad 3-or-4 Finger packet (APD = 0x3):

```
          b7   b6   b5   b4   b3   b2   b1   b0
byte 0:  SWM  SWR  SWL    1    1  AX6  AX5  AX4
byte 1: AX11 AX10  AX9  AX8  AX7  AZ1  AY4  AZ0
byte 2: AY11 AY10  AY9  OVF  AY8  AY7  AY6  AY5
byte 3:    0    0    1    1    1  BX6  BX5  BX4
byte 4: BX11 BX10  BX9  BX8  BX7  BZ1  BY4  BZ0
byte 5: BY11 BY10  BY9    0  BY8  BY7  BY5  BY5
```

OVF: 5th finger detected

Amiga joystick extensions

Amiga 4-joystick parport extension

Parallel port pins:

Pin	Meaning	Pin	Meaning
2	Up1	6	Up2
3	Down1	7	Down2
4	Left1	8	Left2
5	Right1	9	Right2
13	Fire1	11	Fire2
18	Gnd1	18	Gnd2

Amiga digital joystick pinout

Pin	Meaning
1	Up
2	Down
3	Left
4	Right
5	n/c
6	Fire button
7	+5V (50mA)
8	Gnd
9	Thumb button

Amiga mouse pinout

Pin	Meaning
1	V-pulse
2	H-pulse
3	VQ-pulse
4	HQ-pulse
5	Middle button
6	Left button
7	+5V (50mA)
8	Gnd
9	Right button

Amiga analog joystick pinout

Pin	Meaning
1	Top button
2	Top2 button
3	Trigger button
4	Thumb button
5	Analog X
6	n/c
7	+5V (50mA)
8	Gnd
9	Analog Y

Amiga lightpen pinout

Pin	Meaning
1	n/c
2	n/c
3	n/c
4	n/c
5	Touch button
6	/Beamtrigger
7	+5V (50mA)
8	Gnd
9	Stylus button

NAME	rev	ADDR	type	chip	Description
JOY0DAT		00A	R	Denise	Joystick-mouse 0 data (left vert, horiz)
JOY1DAT		00C	R	Denise	Joystick-mouse 1 data (right vert,horiz)

These addresses each read a 16 bit register. These in turn are loaded from the MDAT serial stream and are clocked in on the rising edge of SCLK. MLD output is used to parallel load the external parallel-to-serial converter.This in turn is loaded with the 4 quadrature inputs from each of two game controller ports (8 total) plus 8 miscellaneous control bits which are new for LISA and can be read in upper 8 bits of LISAID.

Register bits are as follows:

- Mouse counter usage (pins 1,3 =Yclock, pins 2,4 =Xclock)

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
JOY0DAT	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0
JOY1DAT	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

0=LEFT CONTROLLER PAIR, 1=RIGHT CONTROLLER PAIR. (4 counters total). The bit usage for both left and right addresses is shown below. Each 6 bit counter (Y7-Y2,X7-X2) is clocked by 2 of the signals input from the mouse serial stream. Starting with first bit received:

Serial	Bit Name	Description
0	M0H	JOY0DAT Horizontal Clock
1	M0HQ	JOY0DAT Horizontal Clock (quadrature)
2	M0V	JOY0DAT Vertical Clock
3	M0VQ	JOY0DAT Vertical Clock (quadrature)
4	M1V	JOY1DAT Horizontal Clock
5	M1VQ	JOY1DAT Horizontal Clock (quadrature)
6	M1V	JOY1DAT Vertical Clock
7	M1VQ	JOY1DAT Vertical Clock (quadrature)

Bits 1 and 0 of each counter (Y1-Y0,X1-X0) may be read to determine the state of the related input signal pair. This allows these pins to double as joystick switch inputs. Joystick switch closures can be deciphered as follows:

Directions	Pin#	Counter bits
Forward	1	Y1 xor Y0 (BIT#09 xor BIT#08)
Left	3	Y1
Back	2	X1 xor X0 (BIT#01 xor BIT#00)
Right	4	X1

NAME	rev	ADDR	type	chip	Description
JOYTEST		036	W	Denise	Write to all 4 joystick-mouse counters at once.

Mouse counter write test data:

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
JOYxDAT	Y7	Y6	Y5	Y4	Y3	Y2	xx	xx	X7	X6	X5	X4	X3	X2	xx	xx
JOYxDAT	Y7	Y6	Y5	Y4	Y3	Y2	xx	xx	X7	X6	X5	X4	X3	X2	xx	xx

NAME	rev	ADDR	type	chip	Description
POT0DAT	h	012	R	Paula	Pot counter data left pair (vert, horiz)
POT1DAT	h	014	R	Paula	Pot counter data right pair (vert,horiz)

These addresses each read a pair of 8 bit pot counters. (4 counters total). The bit assignment for both addresses is shown below. The counters are stopped by signals from 2 controller connectors (left-right) with 2 pins each.

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RIGHT	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0
LEFT	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

CONNECTORS				PAULA
Loc.	Dir.	Sym	pin	pin
RIGHT	Y	RX	9	33
RIGHT	X	RX	5	32
LEFT	Y	LY	9	36
LEFT	X	LX	5	35

With normal (NTSC or PAL) horiz. line rate, the pots will give a full scale (FF) reading with about 500kohms in one frame time. With proportionally faster horiz line times, the counters will count proportionally faster. This should be noted when doing variable beam displays.

NAME	rev	ADDR	type	chip	Description
POTGO		034	W	Paula	Pot port (4 bit) bi-direction and data, and pot counter start.

NAME	rev	ADDR	type	chip	Description
POTINP		016	R	Paula	Pot pin data read

This register controls a 4 bit bi-direction I/O port that shares the same 4 pins as the 4 pot counters above.

BIT#	FUNCTION	DESCRIPTION
15	OUTRY	Output enable for Paula pin 33
14	DATRY	I/O data Paula pin 33
13	OUTRX	Output enable for Paula pin 32
12	DATRX	I/O data Paula pin 32
11	OUTLY	Out put enable for Paula pin 36
10	DATLY	I/O data Paula pin 36
09	OUTLX	Output enable for Paula pin 35
08	DATLX	I/O data Paula pin 35
07-01	X	Not used
00	START	Start pots (dump capacitors,start counters)

Apple Touchpad Driver (appletouch)

appletouch is a Linux kernel driver for the USB touchpad found on post February 2005 and October 2005 Apple Aluminium Powerbooks.

This driver is derived from Johannes Berg's appletrackpad driver [1], but it has been improved in some

[1] http://johannes.sipsolutions.net/PowerBook/touchpad/

areas:

- appletouch is a full kernel driver, no userspace program is necessary
- appletouch can be interfaced with the synaptics X11 driver, in order to have touchpad acceleration, scrolling, etc.

Credits go to Johannes Berg for reverse-engineering the touchpad protocol, Frank Arnold for further improvements, and Alex Harper for some additional information about the inner workings of the touchpad sensors. Michael Hanselmann added support for the October 2005 models.

Usage

In order to use the touchpad in the basic mode, compile the driver and load the module. A new input device will be detected and you will be able to read the mouse data from /dev/input/mice (using gpm, or X11).

In X11, you can configure the touchpad to use the synaptics X11 driver, which will give additional functionalities, like acceleration, scrolling, 2 finger tap for middle button mouse emulation, 3 finger tap for right button mouse emulation, etc. In order to do this, make sure you're using a recent version of the synaptics driver (tested with 0.14.2, available from [2]), and configure a new input device in your X11 configuration file (take a look below for an example). For additional configuration, see the synaptics driver documentation:

```
Section "InputDevice"
        Identifier      "Synaptics Touchpad"
        Driver          "synaptics"
        Option          "SendCoreEvents"        "true"
        Option          "Device"                "/dev/input/mice"
        Option          "Protocol"              "auto-dev"
        Option          "LeftEdge"              "0"
        Option          "RightEdge"             "850"
        Option          "TopEdge"               "0"
        Option          "BottomEdge"            "645"
        Option          "MinSpeed"              "0.4"
        Option          "MaxSpeed"              "1"
        Option          "AccelFactor"           "0.02"
        Option          "FingerLow"             "0"
        Option          "FingerHigh"            "30"
        Option          "MaxTapMove"            "20"
        Option          "MaxTapTime"            "100"
        Option          "HorizScrollDelta"      "0"
        Option          "VertScrollDelta"       "30"
        Option          "SHMConfig"             "on"
EndSection

Section "ServerLayout"
        ...
        InputDevice     "Mouse"
        InputDevice     "Synaptics Touchpad"
...
EndSection
```

Fuzz problems

The touchpad sensors are very sensitive to heat, and will generate a lot of noise when the temperature changes. This is especially true when you power-on the laptop for the first time.

[2] http://web.archive.org/web/*/http://web.telia.com/~u89404340/touchpad/index.html

The appletouch driver tries to handle this noise and auto adapt itself, but it is not perfect. If finger movements are not recognized anymore, try reloading the driver.

You can activate debugging using the 'debug' module parameter. A value of 0 deactivates any debugging, 1 activates tracing of invalid samples, 2 activates full tracing (each sample is being traced):

```
modprobe appletouch debug=1
```

or:

```
echo "1" > /sys/module/appletouch/parameters/debug
```

Intelligent Keyboard (ikbd) Protocol

Introduction

The Atari Corp. Intelligent Keyboard (ikbd) is a general purpose keyboard controller that is flexible enough that it can be used in a variety of products without modification. The keyboard, with its microcontroller, provides a convenient connection point for a mouse and switch-type joysticks. The ikbd processor also maintains a time-of-day clock with one second resolution. The ikbd has been designed to be general enough that it can be used with a variety of new computer products. Product variations in a number of keyswitches, mouse resolution, etc. can be accommodated. The ikbd communicates with the main processor over a high speed bi-directional serial interface. It can function in a variety of modes to facilitate different applications of the keyboard, joysticks, or mouse. Limited use of the controller is possible in applications in which only a unidirectional communications medium is available by carefully designing the default modes.

Keyboard

The keyboard always returns key make/break scan codes. The ikbd generates keyboard scan codes for each key press and release. The key scan make (key closure) codes start at 1, and are defined in Appendix A. For example, the ISO key position in the scan code table should exist even if no keyswitch exists in that position on a particular keyboard. The break code for each key is obtained by ORing 0x80 with the make code.

The special codes 0xF6 through 0xFF are reserved for use as follows:

Code	Command
0xF6	status report
0xF7	absolute mouse position record
0xF8-0xFB	relative mouse position records (lsbs determined by mouse button states)
0xFC	time-of-day
0xFD	joystick report (both sticks)
0xFE	joystick 0 event
0xFF	joystick 1 event

The two shift keys return different scan codes in this mode. The ENTER key and the RETurn key are also distinct.

Mouse

The mouse port should be capable of supporting a mouse with resolution of approximately 200 counts (phase changes or 'clicks') per inch of travel. The mouse should be scanned at a rate that will permit accurate tracking at velocities up to 10 inches per second. The ikbd can report mouse motion in three distinctly different ways. It can report relative motion, absolute motion in a coordinate system maintained

within the ikbd, or by converting mouse motion into keyboard cursor control key equivalents. The mouse buttons can be treated as part of the mouse or as additional keyboard keys.

Relative Position Reporting

In relative position mode, the ikbd will return relative mouse position records whenever a mouse event occurs. A mouse event consists of a mouse button being pressed or released, or motion in either axis exceeding a settable threshold of motion. Regardless of the threshold, all bits of resolution are returned to the host computer. Note that the ikbd may return mouse relative position reports with significantly more than the threshold delta x or y. This may happen since no relative mouse motion events will be generated: (a) while the keyboard has been 'paused' (the event will be stored until keyboard communications is resumed) (b) while any event is being transmitted.

The relative mouse position record is a three byte record of the form (regardless of keyboard mode):

```
%111110xy           ; mouse position record flag
                    ; where y is the right button state
                    ; and x is the left button state
X                   ; delta x as twos complement integer
Y                   ; delta y as twos complement integer
```

Note that the value of the button state bits should be valid even if the MOUSE BUTTON ACTION has set the buttons to act like part of the keyboard. If the accumulated motion before the report packet is generated exceeds the +127...-128 range, the motion is broken into multiple packets. Note that the sign of the delta y reported is a function of the Y origin selected.

Absolute Position reporting

The ikbd can also maintain absolute mouse position. Commands exist for resetting the mouse position, setting X/Y scaling, and interrogating the current mouse position.

Mouse Cursor Key Mode

The ikbd can translate mouse motion into the equivalent cursor keystrokes. The number of mouse clicks per keystroke is independently programmable in each axis. The ikbd internally maintains mouse motion information to the highest resolution available, and merely generates a pair of cursor key events for each multiple of the scale factor. Mouse motion produces the cursor key make code immediately followed by the break code for the appropriate cursor key. The mouse buttons produce scan codes above those normally assigned for the largest envisioned keyboard (i.e. LEFT=0x74 & RIGHT=0x75).

Joystick

Joystick Event Reporting

In this mode, the ikbd generates a record whenever the joystick position is changed (i.e. for each opening or closing of a joystick switch or trigger).

The joystick event record is two bytes of the form:

```
%1111111x           ; Joystick event marker
                    ; where x is Joystick 0 or 1
%x000yyyy           ; where yyyy is the stick position
                    ; and x is the trigger
```

Joystick Interrogation

The current state of the joystick ports may be interrogated at any time in this mode by sending an 'Interrogate Joystick' command to the ikbd.

The ikbd response to joystick interrogation is a three byte report of the form:

```
0xFD                ; joystick report header
%x000yyyy           ; Joystick 0
%x000yyyy           ; Joystick 1
                    ; where x is the trigger
                    ; and yyy is the stick position
```

Joystick Monitoring

A mode is available that devotes nearly all of the keyboard communications time to reporting the state of the joystick ports at a user specifiable rate. It remains in this mode until reset or commanded into another mode. The PAUSE command in this mode not only stop the output but also temporarily stops scanning the joysticks (samples are not queued).

Fire Button Monitoring

A mode is provided to permit monitoring a single input bit at a high rate. In this mode the ikbd monitors the state of the Joystick 1 fire button at the maximum rate permitted by the serial communication channel. The data is packed 8 bits per byte for transmission to the host. The ikbd remains in this mode until reset or commanded into another mode. The PAUSE command in this mode not only stops the output but also temporarily stops scanning the button (samples are not queued).

Joystick Key Code Mode

The ikbd may be commanded to translate the use of either joystick into the equivalent cursor control keystroke(s). The ikbd provides a single breakpoint velocity joystick cursor. Joystick events produce the make code, immediately followed by the break code for the appropriate cursor motion keys. The trigger or fire buttons of the joysticks produce pseudo key scan codes above those used by the largest key matrix envisioned (i.e. JOYSTICK0=0x74, JOYSTICK1=0x75).

Time-of-Day Clock

The ikbd also maintains a time-of-day clock for the system. Commands are available to set and interrogate the timer-of-day clock. Time-keeping is maintained down to a resolution of one second.

Status Inquiries

The current state of ikbd modes and parameters may be found by sending status inquiry commands that correspond to the ikbd set commands.

Power-Up Mode

The keyboard controller will perform a simple self-test on power-up to detect major controller faults (ROM checksum and RAM test) and such things as stuck keys. Any keys down at power-up are presumed to be stuck, and their BREAK (sic) code is returned (which without the preceding MAKE code is a flag for a keyboard error). If the controller self-test completes without error, the code 0xF0 is returned. (This code will be used to indicate the version/release of the ikbd controller. The first release of the ikbd is version 0xF0, should there be a second release it will be 0xF1, and so on.) The ikbd defaults to a mouse position

reporting with threshold of 1 unit in either axis and the Y=0 origin at the top of the screen, and joystick event reporting mode for joystick 1, with both buttons being logically assigned to the mouse. After any joystick command, the ikbd assumes that joysticks are connected to both Joystick0 and Joystick1. Any mouse command (except MOUSE DISABLE) then causes port 0 to again be scanned as if it were a mouse, and both buttons are logically connected to it. If a mouse disable command is received while port 0 is presumed to be a mouse, the button is logically assigned to Joystick1 (until the mouse is reenabled by another mouse command).

ikbd Command Set

This section contains a list of commands that can be sent to the ikbd. Command codes (such as 0x00) which are not specified should perform no operation (NOPs).

RESET

```
0x80
0x01
```

N.B. The RESET command is the only two byte command understood by the ikbd. Any byte following an 0x80 command byte other than 0x01 is ignored (and causes the 0x80 to be ignored). A reset may also be caused by sending a break lasting at least 200mS to the ikbd. Executing the RESET command returns the keyboard to its default (power-up) mode and parameter settings. It does not affect the time-of-day clock. The RESET command or function causes the ikbd to perform a simple self-test. If the test is successful, the ikbd will send the code of 0xF0 within 300mS of receipt of the RESET command (or the end of the break, or power-up). The ikbd will then scan the key matrix for any stuck (closed) keys. Any keys found closed will cause the break scan code to be generated (the break code arriving without being preceded by the make code is a flag for a key matrix error).

SET MOUSE BUTTON ACTION

```
0x07
%00000mss           ; mouse button action
                    ;       (m is presumed = 1 when in MOUSE KEYCODE mode)
                    ; mss=0xy, mouse button press or release causes mouse
                    ;  position report
                    ;  where y=1, mouse key press causes absolute report
                    ;  and x=1, mouse key release causes absolute report
                    ; mss=100, mouse buttons act like keys
```

This command sets how the ikbd should treat the buttons on the mouse. The default mouse button action mode is %00000000, the buttons are treated as part of the mouse logically. When buttons act like keys, LEFT=0x74 & RIGHT=0x75.

SET RELATIVE MOUSE POSITION REPORTING

```
0x08
```

Set relative mouse position reporting. (DEFAULT) Mouse position packets are generated asynchronously by the ikbd whenever motion exceeds the setable threshold in either axis (see SET MOUSE THRESHOLD). Depending upon the mouse key mode, mouse position reports may also be generated when either mouse button is pressed or released. Otherwise the mouse buttons behave as if they were keyboard keys.

SET ABSOLUTE MOUSE POSITIONING

```
0x09
XMSB                ; X maximum (in scaled mouse clicks)
XLSB
YMSB                ; Y maximum (in scaled mouse clicks)
YLSB
```

Set absolute mouse position maintenance. Resets the ikbd maintained X and Y coordinates. In this mode, the value of the internally maintained coordinates does NOT wrap between 0 and large positive numbers. Excess motion below 0 is ignored. The command sets the maximum positive value that can be attained in the scaled coordinate system. Motion beyond that value is also ignored.

SET MOUSE KEYCODE MOSE

```
0x0A
deltax              ; distance in X clicks to return (LEFT) or (RIGHT)
deltay              ; distance in Y clicks to return (UP) or (DOWN)
```

Set mouse monitoring routines to return cursor motion keycodes instead of either RELATIVE or ABSOLUTE motion records. The ikbd returns the appropriate cursor keycode after mouse travel exceeding the user specified deltas in either axis. When the keyboard is in key scan code mode, mouse motion will cause the make code immediately followed by the break code. Note that this command is not affected by the mouse motion origin.

SET MOUSE THRESHOLD

```
0x0B
X                   ; x threshold in mouse ticks (positive integers)
Y                   ; y threshold in mouse ticks (positive integers)
```

This command sets the threshold before a mouse event is generated. Note that it does NOT affect the resolution of the data returned to the host. This command is valid only in RELATIVE MOUSE POSITIONING mode. The thresholds default to 1 at RESET (or power-up).

SET MOUSE SCALE

```
0x0C
X                   ; horizontal mouse ticks per internal X
Y                   ; vertical mouse ticks per internal Y
```

This command sets the scale factor for the ABSOLUTE MOUSE POSITIONING mode. In this mode, the specified number of mouse phase changes ('clicks') must occur before the internally maintained coordinate is changed by one (independently scaled for each axis). Remember that the mouse position information is available only by interrogating the ikbd in the ABSOLUTE MOUSE POSITIONING mode unless the ikbd has been commanded to report on button press or release (see SET MOSE BUTTON ACTION).

INTERROGATE MOUSE POSITION

```
0x0D
Returns:
        0xF7        ; absolute mouse position header
BUTTONS
        0000dcba    ; where a is right button down since last interrogation
                    ; b is right button up since last
```

```
                ; c is left button down since last
                ; d is left button up since last
        XMSB    ; X coordinate
        XLSB
        YMSB    ; Y coordinate
        YLSB
```

The INTERROGATE MOUSE POSITION command is valid when in the ABSOLUTE MOUSE POSITIONING mode, regardless of the setting of the MOUSE BUTTON ACTION.

LOAD MOUSE POSITION

```
0x0E
0x00            ; filler
XMSB            ; X coordinate
XLSB            ; (in scaled coordinate system)
YMSB            ; Y coordinate
YLSB
```

This command allows the user to preset the internally maintained absolute mouse position.

SET Y=0 AT BOTTOM

```
0x0F
```

This command makes the origin of the Y axis to be at the bottom of the logical coordinate system internal to the ikbd for all relative or absolute mouse motion. This causes mouse motion toward the user to be negative in sign and away from the user to be positive.

SET Y=0 AT TOP

```
0x10
```

Makes the origin of the Y axis to be at the top of the logical coordinate system within the ikbd for all relative or absolute mouse motion. (DEFAULT) This causes mouse motion toward the user to be positive in sign and away from the user to be negative.

RESUME

```
0x11
```

Resume sending data to the host. Since any command received by the ikbd after its output has been paused also causes an implicit RESUME this command can be thought of as a NO OPERATION command. If this command is received by the ikbd and it is not PAUSED, it is simply ignored.

DISABLE MOUSE

```
0x12
```

All mouse event reporting is disabled (and scanning may be internally disabled). Any valid mouse mode command resumes mouse motion monitoring. (The valid mouse mode commands are SET RELATIVE MOUSE POSITION REPORTING, SET ABSOLUTE MOUSE POSITIONING, and SET MOUSE KEYCODE MODE.) N.B. If the mouse buttons have been commanded to act like keyboard keys, this command DOES affect their actions.

PAUSE OUTPUT

```
0x13
```

Stop sending data to the host until another valid command is received. Key matrix activity is still monitored and scan codes or ASCII characters enqueued (up to the maximum supported by the microcontroller) to be sent when the host allows the output to be resumed. If in the JOYSTICK EVENT REPORTING mode, joystick events are also queued. Mouse motion should be accumulated while the output is paused. If the ikbd is in RELATIVE MOUSE POSITIONING REPORTING mode, motion is accumulated beyond the normal threshold limits to produce the minimum number of packets necessary for transmission when output is resumed. Pressing or releasing either mouse button causes any accumulated motion to be immediately queued as packets, if the mouse is in RELATIVE MOUSE POSITION REPORTING mode. Because of the limitations of the microcontroller memory this command should be used sparingly, and the output should not be shut of for more than <tbd> milliseconds at a time. The output is stopped only at the end of the current 'even'. If the PAUSE OUTPUT command is received in the middle of a multiple byte report, the packet will still be transmitted to conclusion and then the PAUSE will take effect. When the ikbd is in either the JOYSTICK MONITORING mode or the FIRE BUTTON MONITORING mode, the PAUSE OUTPUT command also temporarily stops the monitoring process (i.e. the samples are not enqueued for transmission).

SET JOYSTICK EVENT REPORTING

```
0x14
```

Enter JOYSTICK EVENT REPORTING mode (DEFAULT). Each opening or closure of a joystick switch or trigger causes a joystick event record to be generated.

SET JOYSTICK INTERROGATION MODE

```
0x15
```

Disables JOYSTICK EVENT REPORTING. Host must send individual JOYSTICK INTERROGATE commands to sense joystick state.

JOYSTICK INTERROGATE

```
0x16
```

Return a record indicating the current state of the joysticks. This command is valid in either the JOYSTICK EVENT REPORTING mode or the JOYSTICK INTERROGATION MODE.

SET JOYSTICK MONITORING

```
0x17
rate                ; time between samples in hundredths of a second
Returns: (in packets of two as long as in mode)
        %000000xy   ; where y is JOYSTICK1 Fire button
                    ; and x is JOYSTICK0 Fire button
        %nnnnmmmm   ; where m is JOYSTICK1 state
                    ; and n is JOYSTICK0 state
```

Sets the ikbd to do nothing but monitor the serial command line, maintain the time-of-day clock, and monitor the joystick. The rate sets the interval between joystick samples. N.B. The user should not set the rate higher than the serial communications channel will allow the 2 bytes packets to be transmitted.

SET FIRE BUTTON MONITORING

```
0x18
Returns: (as long as in mode)
        %bbbbbbbb   ; state of the JOYSTICK1 fire button packed
                    ; 8 bits per byte, the first sample if the MSB
```

Set the ikbd to do nothing but monitor the serial command line, maintain the time-of-day clock, and monitor the fire button on Joystick 1. The fire button is scanned at a rate that causes 8 samples to be made in the time it takes for the previous byte to be sent to the host (i.e. scan rate = 8/10 * baud rate). The sample interval should be as constant as possible.

SET JOYSTICK KEYCODE MODE

```
0x19
RX                  ; length of time (in tenths of seconds) until
                    ; horizontal velocity breakpoint is reached
RY                  ; length of time (in tenths of seconds) until
                    ; vertical velocity breakpoint is reached
TX                  ; length (in tenths of seconds) of joystick closure
                    ; until horizontal cursor key is generated before RX
                    ; has elapsed
TY                  ; length (in tenths of seconds) of joystick closure
                    ; until vertical cursor key is generated before RY
                    ; has elapsed
VX                  ; length (in tenths of seconds) of joystick closure
                    ; until horizontal cursor keystrokes are generated
                    ; after RX has elapsed
VY                  ; length (in tenths of seconds) of joystick closure
                    ; until vertical cursor keystrokes are generated
                    ; after RY has elapsed
```

In this mode, joystick 0 is scanned in a way that simulates cursor keystrokes. On initial closure, a keystroke pair (make/break) is generated. Then up to Rn tenths of seconds later, keystroke pairs are generated every Tn tenths of seconds. After the Rn breakpoint is reached, keystroke pairs are generated every Vn tenths of seconds. This provides a velocity (auto-repeat) breakpoint feature. Note that by setting RX and/or Ry to zero, the velocity feature can be disabled. The values of TX and TY then become meaningless, and the generation of cursor ‘keystrokes’ is set by VX and VY.

DISABLE JOYSTICKS

```
0x1A
```

Disable the generation of any joystick events (and scanning may be internally disabled). Any valid joystick mode command resumes joystick monitoring. (The joystick mode commands are SET JOYSTICK EVENT REPORTING, SET JOYSTICK INTERROGATION MODE, SET JOYSTICK MONITORING, SET FIRE BUTTON MONITORING, and SET JOYSTICK KEYCODE MODE.)

TIME-OF-DAY CLOCK SET

```
0x1B
YY                  ; year (2 least significant digits)
MM                  ; month
DD                  ; day
hh                  ; hour
mm                  ; minute
ss                  ; second
```

All time-of-day data should be sent to the ikbd in packed BCD format. Any digit that is not a valid BCD digit should be treated as a 'don't care' and not alter that particular field of the date or time. This permits setting only some subfields of the time-of-day clock.

INTERROGATE TIME-OF-DAT CLOCK

```
0x1C
Returns:
        0xFC        ; time-of-day event header
        YY          ; year (2 least significant digits)
        MM          ; month
        DD          ; day
        hh          ; hour
        mm          ; minute
        ss          ; second

All time-of-day is sent in packed BCD format.
```

MEMORY LOAD

```
0x20
ADRMSB              ; address in controller
ADRLSB              ; memory to be loaded
NUM                 ; number of bytes (0-128)
{ data }
```

This command permits the host to load arbitrary values into the ikbd controller memory. The time between data bytes must be less than 20ms.

MEMORY READ

```
0x21
ADRMSB              ; address in controller
ADRLSB              ; memory to be read
Returns:
        0xF6        ; status header
        0x20        ; memory access
        { data }    ; 6 data bytes starting at ADR
```

This command permits the host to read from the ikbd controller memory.

CONTROLLER EXECUTE

```
0x22
ADRMSB              ; address of subroutine in
ADRLSB              ; controller memory to be called
```

This command allows the host to command the execution of a subroutine in the ikbd controller memory.

STATUS INQUIRIES

```
Status commands are formed by inclusively ORing 0x80 with the
relevant SET command.

Example:
```

```
0x88 (or 0x89 or 0x8A)  ; request mouse mode
Returns:
        0xF6        ; status response header
        mode        ; 0x08 is RELATIVE
                    ; 0x09 is ABSOLUTE
                    ; 0x0A is KEYCODE
        param1      ; 0 is RELATIVE
                    ; XMSB maximum if ABSOLUTE
                    ; DELTA X is KEYCODE
        param2      ; 0 is RELATIVE
                    ; YMSB maximum if ABSOLUTE
                    ; DELTA Y is KEYCODE
        param3      ; 0 if RELATIVE
                    ; or KEYCODE
                    ; YMSB is ABSOLUTE
        param4      ; 0 if RELATIVE
                    ; or KEYCODE
                    ; YLSB is ABSOLUTE
        0           ; pad
        0
```

The STATUS INQUIRY commands request the ikbd to return either the current mode or the parameters associated with a given command. All status reports are padded to form 8 byte long return packets. The responses to the status requests are designed so that the host may store them away (after stripping off the status report header byte) and later send them back as commands to ikbd to restore its state. The 0 pad bytes will be treated as NOPs by the ikbd.

Valid STATUS INQUIRY commands are:

```
0x87    mouse button action
0x88    mouse mode
0x89
0x8A
0x8B    mnouse threshold
0x8C    mouse scale
0x8F    mouse vertical coordinates
0x90    ( returns       0x0F Y=0 at bottom
                0x10 Y=0 at top )
0x92    mouse enable/disable
        ( returns       0x00 enabled)
                0x12 disabled )
0x94    joystick mode
0x95
0x96
0x9A    joystick enable/disable
        ( returns       0x00 enabled
                0x1A disabled )
```

It is the (host) programmer's responsibility to have only one unanswered inquiry in process at a time. STATUS INQUIRY commands are not valid if the ikbd is in JOYSTICK MONITORING mode or FIRE BUTTON MONITORING mode.

SCAN CODES

The key scan codes returned by the ikbd are chosen to simplify the implementation of GSX.

GSX Standard Keyboard Mapping

Hex	Keytop
01	Esc

Continued on next page

Table 3.1 – continued from previous page

Hex	Keytop
02	1
03	2
04	3
05	4
06	5
07	6
08	7
09	8
0A	9
0B	0
0C	-
0D	=
0E	BS
0F	TAB
10	Q
11	W
12	E
13	R
14	T
15	Y
16	U
17	I
18	O
19	P
1A	[
1B	]
1C	RET
1D	CTRL
1E	A
1F	S
20	D
21	F
22	G
23	H
24	J
25	K
26	L
27	;
28	‘
29	‘
2A	(LEFT) SHIFT
2B	\
2C	Z
2D	X
2E	C
2F	V
30	B
31	N
32	M
33	,
34	.
35	/
36	(RIGHT) SHIFT

Continued on next page

Table 3.1 – continued from previous page

Hex	Keytop
37	{ NOT USED }
38	ALT
39	SPACE BAR
3A	CAPS LOCK
3B	F1
3C	F2
3D	F3
3E	F4
3F	F5
40	F6
41	F7
42	F8
43	F9
44	F10
45	{ NOT USED }
46	{ NOT USED }
47	HOME
48	UP ARROW
49	{ NOT USED }
4A	KEYPAD -
4B	LEFT ARROW
4C	{ NOT USED }
4D	RIGHT ARROW
4E	KEYPAD +
4F	{ NOT USED }
50	DOWN ARROW
51	{ NOT USED }
52	INSERT
53	DEL
54	{ NOT USED }
5F	{ NOT USED }
60	ISO KEY
61	UNDO
62	HELP
63	KEYPAD (
64	KEYPAD /
65	KEYPAD *
66	KEYPAD *
67	KEYPAD 7
68	KEYPAD 8
69	KEYPAD 9
6A	KEYPAD 4
6B	KEYPAD 5
6C	KEYPAD 6
6D	KEYPAD 1
6E	KEYPAD 2
6F	KEYPAD 3
70	KEYPAD 0
71	KEYPAD .
72	KEYPAD ENTER

BCM5974 Driver (bcm5974)

The USB initialization and package decoding was made by Scott Shawcroft as part of the touchd user-space driver project:

The BCM5974 driver is based on the appletouch driver:

This driver adds support for the multi-touch trackpad on the new Apple Macbook Air and Macbook Pro laptops. It replaces the appletouch driver on those computers, and integrates well with the synaptics driver of the Xorg system.

Known to work on Macbook Air, Macbook Pro Penryn and the new unibody Macbook 5 and Macbook Pro 5.

Usage

The driver loads automatically for the supported usb device ids, and becomes available both as an event device (/dev/input/event*) and as a mouse via the mousedev driver (/dev/input/mice).

USB Race

The Apple multi-touch trackpads report both mouse and keyboard events via different interfaces of the same usb device. This creates a race condition with the HID driver, which, if not told otherwise, will find the standard HID mouse and keyboard, and claim the whole device. To remedy, the usb product id must be listed in the mouse_ignore list of the hid driver.

Debug output

To ease the development for new hardware version, verbose packet output can be switched on with the debug kernel module parameter. The range [1-9] yields different levels of verbosity. Example (as root):

```
echo -n 9 > /sys/module/bcm5974/parameters/debug

tail -f /var/log/debug

echo -n 0 > /sys/module/bcm5974/parameters/debug
```

Trivia

The driver was developed at the ubuntu forums in June 2008 [1], and now has a more permanent home at bitmath.org [2].

[1] http://ubuntuforums.org/showthread.php?t=840040
[2] http://bitmath.org/code/

CMA3000-D0x Accelerometer

Supported chips: * VTI CMA3000-D0x

Datasheet: CMA3000-D0X Product Family Specification 8281000A.02.pdf <http://www.vti.fi/en/>

Author Hemanth V <hemanthv@ti.com>

Description

CMA3000 Tri-axis accelerometer supports Motion detect, Measurement and Free fall modes.

Motion Detect Mode: Its the low power mode where interrupts are generated only when motion exceeds the defined thresholds.

Measurement Mode: This mode is used to read the acceleration data on X,Y,Z axis and supports 400, 100, 40 Hz sample frequency.

Free fall Mode: This mode is intended to save system resources.

Threshold values: Chip supports defining threshold values for above modes which includes time and g value. Refer product specifications for more details.

CMA3000 chip supports mutually exclusive I2C and SPI interfaces for communication, currently the driver supports I2C based communication only. Initial configuration for bus mode is set in non volatile memory and can later be modified through bus interface command.

Driver reports acceleration data through input subsystem. It generates ABS_MISC event with value 1 when free fall is detected.

Platform data need to be configured for initial default values.

Platform Data

fuzz_x: Noise on X Axis

fuzz_y: Noise on Y Axis

fuzz_z: Noise on Z Axis

g_range: G range in milli g i.e 2000 or 8000

mode: Default Operating mode

mdthr: Motion detect g range threshold value

mdfftmr: Motion detect and free fall time threshold value

ffthr: Free fall g range threshold value

Input Interface

Input driver version is 1.0.0 Input device ID: bus 0x18 vendor 0x0 product 0x0 version 0x0 Input device name: “cma3000-accelerometer”

Supported events:

```
Event type 0 (Sync)
Event type 3 (Absolute)
  Event code 0 (X)
    Value     47
    Min    -8000
    Max     8000
    Fuzz     200
```

```
  Event code 1 (Y)
    Value    -28
    Min    -8000
    Max     8000
    Fuzz     200
  Event code 2 (Z)
    Value    905
    Min    -8000
    Max     8000
    Fuzz     200
  Event code 40 (Misc)
    Value      0
    Min        0
    Max        1
Event type 4 (Misc)
```

Register/Platform parameters Description

mode:

```
0: power down mode
1: 100 Hz Measurement mode
2: 400 Hz Measurement mode
3: 40 Hz Measurement mode
4: Motion Detect mode (default)
5: 100 Hz Free fall mode
6: 40 Hz Free fall mode
7: Power off mode
```

grange:

```
2000: 2000 mg or 2G Range
8000: 8000 mg or 8G Range
```

mdthr:

```
X: X * 71mg (8G Range)
X: X * 18mg (2G Range)
```

mdfftmr:

```
X: (X & 0x70) * 100 ms (MDTMR)
   (X & 0x0F) * 2.5 ms (FFTMR 400 Hz)
   (X & 0x0F) * 10 ms  (FFTMR 100 Hz)
```

ffthr:

```
X: (X >> 2) * 18mg (2G Range)
X: (X & 0x0F) * 71 mg (8G Range)
```

Crystal SoundFusion CS4610/CS4612/CS461 joystick

This is a new low-level driver to support analog joystick attached to Crystal SoundFusion CS4610/CS4612/CS4615. This code is based upon Vortex/Solo drivers as an example of decoration style, and ALSA 0.5.8a kernel drivers as an chipset documentation and samples.

This version does not have cooked mode support; the basic code is present here, but have not tested completely. The button analysis is completed in this mode, but the axis movement is not.

Raw mode works fine with analog joystick front-end driver and cs461x driver as a backend. I've tested this driver with CS4610, 4-axis and 4-button joystick; I mean the jstest utility. Also I've tried to play in xracer game using joystick, and the result is better than keyboard only mode.

The sensitivity and calibrate quality have not been tested; the two reasons are performed: the same hardware cannot work under Win95 (blue screen in VJOYD); I have no documentation on my chip; and the existing behavior in my case was not raised the requirement of joystick calibration. So the driver have no code to perform hardware related calibration.

This driver have the basic support for PCI devices only; there is no ISA or PnP ISA cards supported.

The driver works with ALSA drivers simultaneously. For example, the xracer uses joystick as input device and PCM device as sound output in one time. There are no sound or input collisions detected. The source code have comments about them; but I've found the joystick can be initialized separately of ALSA modules. So, you can use only one joystick driver without ALSA drivers. The ALSA drivers are not needed to compile or run this driver.

There are no debug information print have been placed in source, and no specific options required to work this driver. The found chipset parameters are printed via printk(KERN_INFO "..."), see the /var/log/messages to inspect cs461x: prefixed messages to determine possible card detection errors.

Regards, Viktor

EDT ft5x06 based Polytouch devices

The edt-ft5x06 driver is useful for the EDT "Polytouch" family of capacitive touch screens. Note that it is *not* suitable for other devices based on the focaltec ft5x06 devices, since they contain vendor-specific firmware. In particular this driver is not suitable for the Nook tablet.

It has been tested with the following devices:

- EP0350M06
- EP0430M06
- EP0570M06
- EP0700M06

The driver allows configuration of the touch screen via a set of sysfs files:

/sys/class/input/eventX/device/device/threshold: allows setting the "click"-threshold in the range from 0 to 80.

/sys/class/input/eventX/device/device/gain: allows setting the sensitivity in the range from 0 to 31. Note that lower values indicate higher sensitivity.

/sys/class/input/eventX/device/device/offset: allows setting the edge compensation in the range from 0 to 31.

/sys/class/input/eventX/device/device/report_rate: allows setting the report rate in the range from 3 to 14.

For debugging purposes the driver provides a few files in the debug filesystem (if available in the kernel). In /sys/kernel/debug/edt_ft5x06 you'll find the following files:

num_x, num_y: (readonly) contains the number of sensor fields in X- and Y-direction.

mode: allows switching the sensor between "factory mode" and "operation mode" by writing "1" or "0" to it. In factory mode (1) it is possible to get the raw data from the sensor. Note that in factory mode regular events don't get delivered and the options described above are unavailable.

raw_data: contains num_x * num_y big endian 16 bit values describing the raw values for each sensor field. Note that each read() call on this files triggers a new readout. It is recommended to provide a buffer big enough to contain num_x * num_y * 2 bytes.

Note that reading raw_data gives a I/O error when the device is not in factory mode. The same happens when reading/writing to the parameter files when the device is not in regular operation mode.

Elantech Touchpad Driver

Copyright (C) 2007-2008 Arjan Opmeer <arjan@opmeer.net>

Extra information for hardware version 1 found and provided by Steve Havelka

Version 2 (EeePC) hardware support based on patches received from Woody at Xandros and forwarded to me by user StewieGriffin at the eeeuser.com forum

Introduction

Currently the Linux Elantech touchpad driver is aware of four different hardware versions unimaginatively called version 1,version 2, version 3 and version 4. Version 1 is found in “older” laptops and uses 4 bytes per packet. Version 2 seems to be introduced with the EeePC and uses 6 bytes per packet, and provides additional features such as position of two fingers, and width of the touch. Hardware version 3 uses 6 bytes per packet (and for 2 fingers the concatenation of two 6 bytes packets) and allows tracking of up to 3 fingers. Hardware version 4 uses 6 bytes per packet, and can combine a status packet with multiple head or motion packets. Hardware version 4 allows tracking up to 5 fingers.

Some Hardware version 3 and version 4 also have a trackpoint which uses a separate packet format. It is also 6 bytes per packet.

The driver tries to support both hardware versions and should be compatible with the Xorg Synaptics touchpad driver and its graphical configuration utilities.

Note that a mouse button is also associated with either the touchpad or the trackpoint when a trackpoint is available. Disabling the Touchpad in xorg (TouchPadOff=0) will also disable the buttons associated with the touchpad.

Additionally the operation of the touchpad can be altered by adjusting the contents of some of its internal registers. These registers are represented by the driver as sysfs entries under /sys/bus/serio/drivers/psmouse/serio? that can be read from and written to.

Currently only the registers for hardware version 1 are somewhat understood. Hardware version 2 seems to use some of the same registers but it is not known whether the bits in the registers represent the same thing or might have changed their meaning.

On top of that, some register settings have effect only when the touchpad is in relative mode and not in absolute mode. As the Linux Elantech touchpad driver always puts the hardware into absolute mode not all information mentioned below can be used immediately. But because there is no freely available Elantech documentation the information is provided here anyway for completeness sake.

Extra knobs

Currently the Linux Elantech touchpad driver provides three extra knobs under /sys/bus/serio/drivers/psmouse/serio? for the user.

- debug

  Turn different levels of debugging ON or OFF.

  By echoing “0” to this file all debugging will be turned OFF.

  Currently a value of “1” will turn on some basic debugging and a value of “2” will turn on packet debugging. For hardware version 1 the default is OFF. For version 2 the default is “1”.

Turning packet debugging on will make the driver dump every packet received to the syslog before processing it. Be warned that this can generate quite a lot of data!

- paritycheck

  Turns parity checking ON or OFF.

  By echoing "0" to this file parity checking will be turned OFF. Any non-zero value will turn it ON. For hardware version 1 the default is ON. For version 2 the default it is OFF.

  Hardware version 1 provides basic data integrity verification by calculating a parity bit for the last 3 bytes of each packet. The driver can check these bits and reject any packet that appears corrupted. Using this knob you can bypass that check.

  Hardware version 2 does not provide the same parity bits. Only some basic data consistency checking can be done. For now checking is disabled by default. Currently even turning it on will do nothing.

- crc_enabled

  Sets crc_enabled to 0/1. The name "crc_enabled" is the official name of this integrity check, even though it is not an actual cyclic redundancy check.

  Depending on the state of crc_enabled, certain basic data integrity verification is done by the driver on hardware version 3 and 4. The driver will reject any packet that appears corrupted. Using this knob, The state of crc_enabled can be altered with this knob.

  Reading the crc_enabled value will show the active value. Echoing "0" or "1" to this file will set the state to "0" or "1".

Differentiating hardware versions

To detect the hardware version, read the version number as param[0].param[1].param[2]:

```
4 bytes version: (after the arrow is the name given in the Dell-provided driver)
02.00.22 => EF013
02.06.00 => EF019
```

In the wild, there appear to be more versions, such as 00.01.64, 01.00.21, 02.00.00, 02.00.04, 02.00.06:

```
6 bytes:
02.00.30 => EF113
02.08.00 => EF023
02.08.XX => EF123
02.0B.00 => EF215
04.01.XX => Scroll_EF051
04.02.XX => EF051
```

In the wild, there appear to be more versions, such as 04.03.01, 04.04.11. There appears to be almost no difference, except for EF113, which does not report pressure/width and has different data consistency checks.

Probably all the versions with param[0] <= 01 can be considered as 4 bytes/firmware 1. The versions < 02.08.00, with the exception of 02.00.30, as 4 bytes/firmware 2. Everything >= 02.08.00 can be considered as 6 bytes.

Hardware version 1

Registers

By echoing a hexadecimal value to a register it contents can be altered.

For example:

```
echo -n 0x16 > reg_10
```

- reg_10:

```
bit   7   6   5   4   3   2   1   0
      B   C   T   D   L   A   S   E

      E: 1 = enable smart edges unconditionally
      S: 1 = enable smart edges only when dragging
      A: 1 = absolute mode (needs 4 byte packets, see reg_11)
      L: 1 = enable drag lock (see reg_22)
      D: 1 = disable dynamic resolution
      T: 1 = disable tapping
      C: 1 = enable corner tap
      B: 1 = swap left and right button
```

- reg_11:

```
bit   7   6   5   4   3   2   1   0
      1   0   0   H   V   1   F   P

      P: 1 = enable parity checking for relative mode
      F: 1 = enable native 4 byte packet mode
      V: 1 = enable vertical scroll area
      H: 1 = enable horizontal scroll area
```

- reg_20:

```
single finger width?
```

- reg_21:

```
scroll area width (small: 0x40 ... wide: 0xff)
```

- reg_22:

```
drag lock time out (short: 0x14 ... long: 0xfe;
                    0xff = tap again to release)
```

- reg_23:

```
tap make timeout?
```

- reg_24:

```
tap release timeout?
```

- reg_25:

```
smart edge cursor speed (0x02 = slow, 0x03 = medium, 0x04 = fast)
```

- reg_26:

```
smart edge activation area width?
```

Native relative mode 4 byte packet format

byte 0:

```
bit   7   6   5   4   3   2   1   0
      c   c  p2  p1   1   M   R   L

      L, R, M = 1 when Left, Right, Middle mouse button pressed
         some models have M as byte 3 odd parity bit
      when parity checking is enabled (reg_11, P = 1):
         p1..p2 = byte 1 and 2 odd parity bit
      c = 1 when corner tap detected
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
     dx7 dx6 dx5 dx4 dx3 dx2 dx1 dx0

      dx7..dx0 = x movement;   positive = right, negative = left
      byte 1 = 0xf0 when corner tap detected
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
     dy7 dy6 dy5 dy4 dy3 dy2 dy1 dy0

      dy7..dy0 = y movement;   positive = up,    negative = down
```

byte 3:

```
parity checking enabled (reg_11, P = 1):

   bit   7   6   5   4   3   2   1   0
         w   h  n1  n0  ds3 ds2 ds1 ds0

         normally:
            ds3..ds0 = scroll wheel amount and direction
                       positive = down or left
                       negative = up or right
         when corner tap detected:
            ds0 = 1 when top right corner tapped
            ds1 = 1 when bottom right corner tapped
            ds2 = 1 when bottom left corner tapped
            ds3 = 1 when top left corner tapped
         n1..n0 = number of fingers on touchpad
            only models with firmware 2.x report this, models with
            firmware 1.x seem to map one, two and three finger taps
            directly to L, M and R mouse buttons
         h = 1 when horizontal scroll action
         w = 1 when wide finger touch?

otherwise (reg_11, P = 0):

   bit   7   6   5   4   3   2   1   0
        ds7 ds6 ds5 ds4 ds3 ds2 ds1 ds0

         ds7..ds0 = vertical scroll amount and direction
                    negative = up
                    positive = down
```

Native absolute mode 4 byte packet format

EF013 and EF019 have a special behaviour (due to a bug in the firmware?), and when 1 finger is touching, the first 2 position reports must be discarded. This counting is reset whenever a different number of fingers is reported.

byte 0:

```
firmware version 1.x:

   bit   7   6   5   4   3   2   1   0
         D   U  p1  p2   1  p3   R   L

         L, R = 1 when Left, Right mouse button pressed
         p1..p3 = byte 1..3 odd parity bit
         D, U = 1 when rocker switch pressed Up, Down

firmware version 2.x:

   bit   7   6   5   4   3   2   1   0
        n1  n0  p2  p1   1  p3   R   L

         L, R = 1 when Left, Right mouse button pressed
         p1..p3 = byte 1..3 odd parity bit
         n1..n0 = number of fingers on touchpad
```

byte 1:

```
firmware version 1.x:

   bit   7   6   5   4   3   2   1   0
         f   0  th  tw  x9  x8  y9  y8

         tw = 1 when two finger touch
         th = 1 when three finger touch
         f  = 1 when finger touch

firmware version 2.x:

   bit   7   6   5   4   3   2   1   0
         .   .   .   .  x9  x8  y9  y8
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
     x7  x6  x5  x4  x3  x2  x1  x0

      x9..x0 = absolute x value (horizontal)
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
     y7  y6  y5  y4  y3  y2  y1  y0

      y9..y0 = absolute y value (vertical)
```

Hardware version 2

Registers

By echoing a hexadecimal value to a register it contents can be altered.

For example:

```
echo -n 0x56 > reg_10
```

- reg_10:

```
bit   7   6   5   4   3   2   1   0
      0   1   0   1   0   1   D   0

      D: 1 = enable drag and drop
```

- reg_11:

```
bit   7   6   5   4   3   2   1   0
      1   0   0   0   S   0   1   0

      S: 1 = enable vertical scroll
```

- reg_21:

```
unknown (0x00)
```

- reg_22:

```
drag and drop release time out (short: 0x70 ... long 0x7e;
                          0x7f = never i.e. tap again to release)
```

Native absolute mode 6 byte packet format

Parity checking and packet re-synchronization

There is no parity checking, however some consistency checks can be performed.

For instance for EF113:

```
SA1= packet[0];
A1 = packet[1];
B1 = packet[2];
SB1= packet[3];
C1 = packet[4];
D1 = packet[5];
if( (((SA1 & 0x3C) != 0x3C) && ((SA1 & 0xC0) != 0x80)) || // check Byte 1
    (((SA1 & 0x0C) != 0x0C) && ((SA1 & 0xC0) == 0x80)) || // check Byte 1 (one finger pressed)
    (((SA1 & 0xC0) != 0x80) && (( A1 & 0xF0) != 0x00)) || // check Byte 2
    (((SB1 & 0x3E) != 0x38) && ((SA1 & 0xC0) != 0x80)) || // check Byte 4
    (((SB1 & 0x0E) != 0x08) && ((SA1 & 0xC0) == 0x80)) || // check Byte 4 (one finger pressed)
    (((SA1 & 0xC0) != 0x80) && (( C1 & 0xF0) != 0x00))  ) // check Byte 5
        // error detected
```

For all the other ones, there are just a few constant bits:

```
if( ((packet[0] & 0x0C) != 0x04) ||
    ((packet[3] & 0x0f) != 0x02) )
        // error detected
```

In case an error is detected, all the packets are shifted by one (and packet[0] is discarded).

One/Three finger touch

byte 0:

```
bit   7   6   5   4   3   2   1   0
      n1  n0  w3  w2   .   .   R   L
```

```
      L, R = 1 when Left, Right mouse button pressed
      n1..n0 = number of fingers on touchpad
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
      p7  p6  p5  p4 x11 x10 x9  x8
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
      x7  x6  x5  x4  x3  x2  x1  x0

      x11..x0 = absolute x value (horizontal)
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
      n4  vf  w1  w0   .   .   .  b2

      n4 = set if more than 3 fingers (only in 3 fingers mode)
      vf = a kind of flag ? (only on EF123, 0 when finger is over one
           of the buttons, 1 otherwise)
      w3..w0 = width of the finger touch (not EF113)
      b2 (on EF113 only, 0 otherwise), b2.R.L indicates one button pressed:
             0 = none
             1 = Left
             2 = Right
             3 = Middle (Left and Right)
             4 = Forward
             5 = Back
             6 = Another one
             7 = Another one
```

byte 4:

```
bit   7   6   5   4   3   2   1   0
     p3  p1  p2  p0  y11 y10 y9  y8

      p7..p0 = pressure (not EF113)
```

byte 5:

```
bit   7   6   5   4   3   2   1   0
     y7  y6  y5  y4  y3  y2  y1  y0

      y11..y0 = absolute y value (vertical)
```

Two finger touch

Note that the two pairs of coordinates are not exactly the coordinates of the two fingers, but only the pair of the lower-left and upper-right coordinates. So the actual fingers might be situated on the other diagonal of the square defined by these two points.

byte 0:

```
bit   7   6   5   4   3   2   1   0
     n1  n0  ay8 ax8  .   .   R   L

      L, R = 1 when Left, Right mouse button pressed
      n1..n0 = number of fingers on touchpad
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
     ax7 ax6 ax5 ax4 ax3 ax2 ax1 ax0

      ax8..ax0 = lower-left finger absolute x value
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
     ay7 ay6 ay5 ay4 ay3 ay2 ay1 ay0

      ay8..ay0 = lower-left finger absolute y value
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
      .   .  by8 bx8  .   .   .   .
```

byte 4:

```
bit   7   6   5   4   3   2   1   0
     bx7 bx6 bx5 bx4 bx3 bx2 bx1 bx0

      bx8..bx0 = upper-right finger absolute x value
```

byte 5:

```
bit   7   6   5   4   3   2   1   0
     by7 by8 by5 by4 by3 by2 by1 by0

      by8..by0 = upper-right finger absolute y value
```

Hardware version 3

Registers

- reg_10:

```
bit   7   6   5   4   3   2   1   0
      0   0   0   0   R   F   T   A

      A: 1 = enable absolute tracking
      T: 1 = enable two finger mode auto correct
      F: 1 = disable ABS Position Filter
      R: 1 = enable real hardware resolution
```

Native absolute mode 6 byte packet format

1 and 3 finger touch shares the same 6-byte packet format, except that 3 finger touch only reports the position of the center of all three fingers.

Firmware would send 12 bytes of data for 2 finger touch.

Note on debounce: In case the box has unstable power supply or other electricity issues, or when number of finger changes, F/W would send “debounce packet” to inform driver that the hardware is in debounce status. The debouce packet has the following signature:

```
byte 0: 0xc4
byte 1: 0xff
byte 2: 0xff
byte 3: 0x02
byte 4: 0xff
byte 5: 0xff
```

When we encounter this kind of packet, we just ignore it.

One/Three finger touch

byte 0:

```
bit   7   6   5   4   3   2   1   0
     n1  n0  w3  w2   0   1   R   L

     L, R = 1 when Left, Right mouse button pressed
     n1..n0 = number of fingers on touchpad
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
     p7  p6  p5  p4 x11 x10  x9  x8
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
     x7  x6  x5  x4  x3  x2  x1  x0

     x11..x0 = absolute x value (horizontal)
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
      0   0  w1  w0   0   0   1   0

      w3..w0 = width of the finger touch
```

byte 4:

```
bit   7   6   5   4   3   2   1   0
     p3  p1  p2  p0  y11 y10 y9  y8

     p7..p0 = pressure
```

byte 5:

```
bit   7   6   5   4   3   2   1   0
     y7  y6  y5  y4  y3  y2  y1  y0

     y11..y0 = absolute y value (vertical)
```

Two finger touch

The packet format is exactly the same for two finger touch, except the hardware sends two 6 byte packets. The first packet contains data for the first finger, the second packet has data for the second finger. So for two finger touch a total of 12 bytes are sent.

Hardware version 4

Registers

- reg_07:

```
bit   7   6   5   4   3   2   1   0
      0   0   0   0   0   0   0   A

      A: 1 = enable absolute tracking
```

Native absolute mode 6 byte packet format

v4 hardware is a true multitouch touchpad, capable of tracking up to 5 fingers. Unfortunately, due to PS/2's limited bandwidth, its packet format is rather complex.

Whenever the numbers or identities of the fingers changes, the hardware sends a status packet to indicate how many and which fingers is on touchpad, followed by head packets or motion packets. A head packet contains data of finger id, finger position (absolute x, y values), width, and pressure. A motion packet contains two fingers' position delta.

For example, when status packet tells there are 2 fingers on touchpad, then we can expect two following head packets. If the finger status doesn't change, the following packets would be motion packets, only sending delta of finger position, until we receive a status packet.

One exception is one finger touch. when a status packet tells us there is only one finger, the hardware would just send head packets afterwards.

Status packet

byte 0:

```
bit   7   6   5   4   3   2   1   0
      .   .   .   .   0   1   R   L

      L, R = 1 when Left, Right mouse button pressed
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
      .   .   . ft4 ft3 ft2 ft1 ft0

      ft4 ft3 ft2 ft1 ft0 ftn = 1 when finger n is on touchpad
```

byte 2:

```
not used
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
      .   .   .   1   0   0   0   0

      constant bits
```

byte 4:

```
bit   7   6   5   4   3   2   1   0
      p   .   .   .   .   .   .   .
```

```
     p = 1 for palm
```

byte 5:

```
not used
```

Head packet

byte 0:

```
bit   7   6   5   4   3   2   1   0
     w3  w2  w1  w0   0   1   R   L

     L, R = 1 when Left, Right mouse button pressed
     w3..w0 = finger width (spans how many trace lines)
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
     p7  p6  p5  p4 x11 x10  x9  x8
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
     x7  x6  x5  x4  x3  x2  x1  x0

     x11..x0 = absolute x value (horizontal)
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
    id2 id1 id0   1   0   0   0   1

    id2..id0 = finger id
```

byte 4:

```
bit   7   6   5   4   3   2   1   0
     p3  p1  p2  p0  y11 y10 y9  y8

     p7..p0 = pressure
```

byte 5:

```
bit   7   6   5   4   3   2   1   0
     y7  y6  y5  y4  y3  y2  y1  y0

     y11..y0 = absolute y value (vertical)
```

Motion packet

byte 0:

```
bit   7   6   5   4   3   2   1   0
    id2 id1 id0   w   0   1   R   L

    L, R = 1 when Left, Right mouse button pressed
```

```
    id2..id0 = finger id
    w = 1 when delta overflows (> 127 or < -128), in this case
    firmware sends us (delta x / 5) and (delta y  / 5)
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
     x7  x6  x5  x4  x3  x2  x1  x0

     x7..x0 = delta x (two's complement)
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
     y7  y6  y5  y4  y3  y2  y1  y0

     y7..y0 = delta y (two's complement)
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
    id2 id1 id0   1   0   0   1   0

    id2..id0 = finger id
```

byte 4:

```
bit   7   6   5   4   3   2   1   0
     x7  x6  x5  x4  x3  x2  x1  x0

     x7..x0 = delta x (two's complement)
```

byte 5:

```
bit   7   6   5   4   3   2   1   0
     y7  y6  y5  y4  y3  y2  y1  y0

     y7..y0 = delta y (two's complement)

     byte 0 ~ 2 for one finger
     byte 3 ~ 5 for another
```

Trackpoint (for Hardware version 3 and 4)

Registers

No special registers have been identified.

Native relative mode 6 byte packet format

Status Packet

byte 0:

```
bit   7   6   5   4   3   2   1   0
      0   0  sx  sy   0   M   R   L
```

byte 1:

```
bit   7   6   5   4   3   2   1   0
    ~sx   0   0   0   0   0   0   0
```

byte 2:

```
bit   7   6   5   4   3   2   1   0
    ~sy   0   0   0   0   0   0   0
```

byte 3:

```
bit   7   6   5   4   3   2   1   0
      0   0 ~sy ~sx   0   1   1   0
```

byte 4:

```
bit   7   6   5   4   3   2   1   0
     x7  x6  x5  x4  x3  x2  x1  x0
```

byte 5:

```
bit   7   6   5   4   3   2   1   0
     y7  y6  y5  y4  y3  y2  y1  y0


      x and y are written in two's complement spread
          over 9 bits with sx/sy the relative top bit and
          x7..x0 and y7..y0 the lower bits.
      ~sx is the inverse of sx, ~sy is the inverse of sy.
      The sign of y is opposite to what the input driver
          expects for a relative movement
```

Driver for tilt-switches connected via GPIOs

Generic driver to read data from tilt switches connected via gpios. Orientation can be provided by one or more than one tilt switches, i.e. each tilt switch providing one axis, and the number of axes is also not limited.

Data structures

The array of struct gpio in the gpios field is used to list the gpios that represent the current tilt state.

The array of struct gpio_tilt_axis describes the axes that are reported to the input system. The values set therein are used for the input_set_abs_params calls needed to init the axes.

The array of struct gpio_tilt_state maps gpio states to the corresponding values to report. The gpio state is represented as a bitfield where the bit-index corresponds to the index of the gpio in the struct gpio array. In the same manner the values stored in the axes array correspond to the elements of the gpio_tilt_axis-array.

Example

Example configuration for a single TS1003 tilt switch that rotates around one axis in 4 steps and emits the current tilt via two GPIOs:

```
static int sg060_tilt_enable(struct device *dev) {
        /* code to enable the sensors */
};
```

```
static void sg060_tilt_disable(struct device *dev) {
        /* code to disable the sensors */
};

static struct gpio sg060_tilt_gpios[] = {
        { SG060_TILT_GPIO_SENSOR1, GPIOF_IN, "tilt_sensor1" },
        { SG060_TILT_GPIO_SENSOR2, GPIOF_IN, "tilt_sensor2" },
};

static struct gpio_tilt_state sg060_tilt_states[] = {
        {
                .gpios = (0 << 1) | (0 << 0),
                .axes = (int[]) {
                        0,
                },
        }, {
                .gpios = (0 << 1) | (1 << 0),
                .axes = (int[]) {
                        1, /* 90 degrees */
                },
        }, {
                .gpios = (1 << 1) | (1 << 0),
                .axes = (int[]) {
                        2, /* 180 degrees */
                },
        }, {
                .gpios = (1 << 1) | (0 << 0),
                .axes = (int[]) {
                        3, /* 270 degrees */
                },
        },
};

static struct gpio_tilt_axis sg060_tilt_axes[] = {
        {
                .axis = ABS_RY,
                .min = 0,
                .max = 3,
                .fuzz = 0,
                .flat = 0,
        },
};

static struct gpio_tilt_platform_data sg060_tilt_pdata= {
        .gpios = sg060_tilt_gpios,
        .nr_gpios = ARRAY_SIZE(sg060_tilt_gpios),

        .axes = sg060_tilt_axes,
        .nr_axes = ARRAY_SIZE(sg060_tilt_axes),

        .states = sg060_tilt_states,
        .nr_states = ARRAY_SIZE(sg060_tilt_states),

        .debounce_interval = 100,

        .poll_interval = 1000,
        .enable = sg060_tilt_enable,
        .disable = sg060_tilt_disable,
};

static struct platform_device sg060_device_tilt = {
        .name = "gpio-tilt-polled",
```

```
        .id = -1,
        .dev = {
                .platform_data = &sg060_tilt_pdata,
        },
};
```

Iforce Protocol

Author Johann Deneux <johann.deneux@gmail.com>

Home page at http://web.archive.org/web/*/http://www.esil.univ-mrs.fr

Additions by Vojtech Pavlik.

Introduction

This document describes what I managed to discover about the protocol used to specify force effects to I-Force 2.0 devices. None of this information comes from Immerse. That's why you should not trust what is written in this document. This document is intended to help understanding the protocol. This is not a reference. Comments and corrections are welcome. To contact me, send an email to: johann.deneux@gmail.com

Warning:

I shall not be held responsible for any damage or harm caused if you try to send data to your I-Force device based on what you read in this document.

Preliminary Notes

All values are hexadecimal with big-endian encoding (msb on the left). Beware, values inside packets are encoded using little-endian. Bytes whose roles are unknown are marked ??? Information that needs deeper inspection is marked (?)

General form of a packet

This is how packets look when the device uses the rs232 to communicate.

2B	OP	LEN	DATA	CS

CS is the checksum. It is equal to the exclusive or of all bytes.

When using USB:

OP	DATA

The 2B, LEN and CS fields have disappeared, probably because USB handles frames and data corruption is handled or unsignificant.

First, I describe effects that are sent by the device to the computer

Device input state

This packet is used to indicate the state of each button and the value of each axis:

```
OP= 01 for a joystick, 03 for a wheel
LEN= Varies from device to device
00 X-Axis lsb
01 X-Axis msb
02 Y-Axis lsb, or gas pedal for a wheel
03 Y-Axis msb, or brake pedal for a wheel
04 Throttle
05 Buttons
06 Lower 4 bits: Buttons
   Upper 4 bits: Hat
07 Rudder
```

Device effects states

```
OP= 02
LEN= Varies
00 ? Bit 1 (Value 2) is the value of the deadman switch
01 Bit 8 is set if the effect is playing. Bits 0 to 7 are the effect id.
02 ??
03 Address of parameter block changed (lsb)
04 Address of parameter block changed (msb)
05 Address of second parameter block changed (lsb)
... depending on the number of parameter blocks updated
```

Force effect

```
OP=  01
LEN= 0e
00 Channel (when playing several effects at the same time, each must
            be assigned a channel)
01 Wave form
        Val 00 Constant
        Val 20 Square
        Val 21 Triangle
        Val 22 Sine
        Val 23 Sawtooth up
        Val 24 Sawtooth down
        Val 40 Spring (Force = f(pos))
        Val 41 Friction (Force = f(velocity)) and Inertia
               (Force = f(acceleration))


02 Axes affected and trigger
        Bits 4-7: Val 2 = effect along one axis. Byte 05 indicates direction
                  Val 4 = X axis only. Byte 05 must contain 5a
                  Val 8 = Y axis only. Byte 05 must contain b4
                  Val c = X and Y axes. Bytes 05 must contain 60
        Bits 0-3: Val 0 = No trigger
                  Val x+1 = Button x triggers the effect
        When the whole byte is 0, cancel the previously set trigger

03-04 Duration of effect (little endian encoding, in ms)

05 Direction of effect, if applicable. Else, see 02 for value to assign.

06-07 Minimum time between triggering.

08-09 Address of periodicity or magnitude parameters
0a-0b Address of attack and fade parameters, or ffff if none.
```

```
*or*
08-09 Address of interactive parameters for X-axis,
      or ffff if not applicable
0a-0b Address of interactive parameters for Y-axis,
      or ffff if not applicable

0c-0d Delay before execution of effect (little endian encoding, in ms)
```

Time based parameters

Attack and fade

```
OP=  02
LEN= 08
00-01 Address where to store the parameters
02-03 Duration of attack (little endian encoding, in ms)
04 Level at end of attack. Signed byte.
05-06 Duration of fade.
07 Level at end of fade.
```

Magnitude

```
OP=  03
LEN= 03
00-01 Address
02 Level. Signed byte.
```

Periodicity

```
OP=  04
LEN= 07
00-01 Address
02 Magnitude. Signed byte.
03 Offset. Signed byte.
04 Phase. Val 00 = 0 deg, Val 40 = 90 degs.
05-06 Period (little endian encoding, in ms)
```

Interactive parameters

```
OP=  05
LEN= 0a
00-01 Address
02 Positive Coeff
03 Negative Coeff
04+05 Offset (center)
06+07 Dead band (Val 01F4 = 5000 (decimal))
08 Positive saturation (Val 0a = 1000 (decimal) Val 64 = 10000 (decimal))
09 Negative saturation
```

The encoding is a bit funny here: For coeffs, these are signed values. The maximum value is 64 (100 decimal), the min is 9c. For the offset, the minimum value is FE0C, the maximum value is 01F4. For the deadband, the minimum value is 0, the max is 03E8.

Controls

```
OP=  41
LEN= 03
00 Channel
01 Start/Stop
        Val 00: Stop
        Val 01: Start and play once.
        Val 41: Start and play n times (See byte 02 below)
02 Number of iterations n.
```

Init

Querying features

```
OP=  ff
Query command. Length varies according to the query type.
The general format of this packet is:
ff 01 QUERY [INDEX] CHECKSUM
responses are of the same form:
FF LEN QUERY VALUE_QUERIED CHECKSUM2
where LEN = 1 + length(VALUE_QUERIED)
```

Query ram size

```
QUERY = 42 ('B'uffer size)
```

The device should reply with the same packet plus two additional bytes containing the size of the memory: ff 03 42 03 e8 CS would mean that the device has 1000 bytes of ram available.

Query number of effects

```
QUERY = 4e ('N'umber of effects)
```

The device should respond by sending the number of effects that can be played at the same time (one byte) ff 02 4e 14 CS would stand for 20 effects.

Vendor's id

```
QUERY = 4d ('M'anufacturer)
```

Query the vendors'id (2 bytes)

Product id

```
QUERY = 50 ('P'roduct)
```

Query the product id (2 bytes)

Open device

```
QUERY = 4f ('O'pen)
```

No data returned.

Close device

```
QUERY = 43 ('C')lose
```

No data returned.

Query effect

```
QUERY = 45 ('E')
```

Send effect type. Returns nonzero if supported (2 bytes)

Firmware Version

```
QUERY = 56 ('V'ersion)
```

Sends back 3 bytes - major, minor, subminor

Initialisation of the device

Set Control

Note:

Device dependent, can be different on different models!

```
OP=  40 <idx> <val> [<val>]
LEN= 2 or 3
00 Idx
   Idx 00 Set dead zone (0..2048)
   Idx 01 Ignore Deadman sensor (0..1)
   Idx 02 Enable comm watchdog (0..1)
   Idx 03 Set the strength of the spring (0..100)
   Idx 04 Enable or disable the spring (0/1)
   Idx 05 Set axis saturation threshold (0..2048)
```

Set Effect State

```
OP=  42 <val>
LEN= 1
00 State
   Bit 3 Pause force feedback
   Bit 2 Enable force feedback
   Bit 0 Stop all effects
```

Set overall

```
OP=  43 <val>
LEN= 1
00 Gain
   Val 00 = 0%
   Val 40 = 50%
   Val 80 = 100%
```

Parameter memory

Each device has a certain amount of memory to store parameters of effects. The amount of RAM may vary, I encountered values from 200 to 1000 bytes. Below is the amount of memory apparently needed for every set of parameters:

- period : 0c
- magnitude : 02
- attack and fade : 0e
- interactive : 08

Appendix: How to study the protocol?

1. Generate effects using the force editor provided with the DirectX SDK, or use Immersion Studio (freely available at their web site in the developer section: www.immersion.com) 2. Start a soft spying RS232 or USB (depending on where you connected your joystick/wheel). I used ComPortSpy from fCoder (alpha version!) 3. Play the effect, and watch what happens on the spy screen.

A few words about ComPortSpy: At first glance, this software seems, hum, well... buggy. In fact, data appear with a few seconds latency. Personally, I restart it every time I play an effect. Remember it's free (as in free beer) and alpha!

URLS

Check http://www.immerse.com for Immersion Studio, and http://www.fcoder.com for ComPortSpy.

I-Force is trademark of Immersion Corp.

Parallel Port Joystick Drivers

Sponsored by SuSE

Disclaimer

Any information in this file is provided as-is, without any guarantee that it will be true. So, use it at your own risk. The possible damages that can happen include burning your parallel port, and/or the sticks and joystick and maybe even more. Like when a lightning kills you it is not our problem.

Introduction

The joystick parport drivers are used for joysticks and gamepads not originally designed for PCs and other computers Linux runs on. Because of that, PCs usually lack the right ports to connect these devices to. Parallel port, because of its ability to change single bits at will, and providing both output and input bits is the most suitable port on the PC for connecting such devices.

Devices supported

Many console and 8-bit computer gamepads and joysticks are supported. The following subsections discuss usage of each.

NES and SNES

The Nintendo Entertainment System and Super Nintendo Entertainment System gamepads are widely available, and easy to get. Also, they are quite easy to connect to a PC, and don't need much processing speed (108 us for NES and 165 us for SNES, compared to about 1000 us for PC gamepads) to communicate with them.

All NES and SNES use the same synchronous serial protocol, clocked from the computer's side (and thus timing insensitive). To allow up to 5 NES and/or SNES gamepads and/or SNES mice connected to the parallel port at once, the output lines of the parallel port are shared, while one of 5 available input lines is assigned to each gamepad.

This protocol is handled by the gamecon.c driver, so that's the one you'll use for NES, SNES gamepads and SNES mice.

The main problem with PC parallel ports is that they don't have +5V power source on any of their pins. So, if you want a reliable source of power for your pads, use either keyboard or joystick port, and make a pass-through cable. You can also pull the power directly from the power supply (the red wire is +5V).

If you want to use the parallel port only, you can take the power is from some data pin. For most gamepad and parport implementations only one pin is needed, and I'd recommend pin 9 for that, the highest data bit. On the other hand, if you are not planning to use anything else than NES / SNES on the port, anything between and including pin 4 and pin 9 will work:

```
(pin 9) -----> Power
```

Unfortunately, there are pads that need a lot more of power, and parallel ports that can't give much current through the data pins. If this is your case, you'll need to use diodes (as a prevention of destroying your parallel port), and combine the currents of two or more data bits together:

```
          Diodes
(pin 9) ----|>|-------+------> Power
                      |
(pin 8) ----|>|-------+
                      |
(pin 7) ----|>|-------+
                      |
<and so on>           :
                      |
(pin 4) ----|>|-------+
```

Ground is quite easy. On PC's parallel port the ground is on any of the pins from pin 18 to pin 25. So use any pin of these you like for the ground:

```
(pin 18) -----> Ground
```

NES and SNES pads have two input bits, Clock and Latch, which drive the serial transfer. These are connected to pins 2 and 3 of the parallel port, respectively:

```
(pin 2) -----> Clock
(pin 3) -----> Latch
```

And the last thing is the NES / SNES data wire. Only that isn't shared and each pad needs its own data pin. The parallel port pins are:

```
(pin 10) -----> Pad 1 data
(pin 11) -----> Pad 2 data
(pin 12) -----> Pad 3 data
(pin 13) -----> Pad 4 data
(pin 15) -----> Pad 5 data
```

Note that pin 14 is not used, since it is not an input pin on the parallel port.

This is everything you need on the PC's side of the connection, now on to the gamepads side. The NES and SNES have different connectors. Also, there are quite a lot of NES clones, and because Nintendo used proprietary connectors for their machines, the cloners couldn't and used standard D-Cannon connectors. Anyway, if you've got a gamepad, and it has buttons A, B, Turbo A, Turbo B, Select and Start, and is connected through 5 wires, then it is either a NES or NES clone and will work with this connection. SNES gamepads also use 5 wires, but have more buttons. They will work as well, of course:

```
Pinout for NES gamepads

          +----> Power
          |
 5 +---------+  7
   | x  x  o   \
   | o  o  o  o |
 4 +------------+ 1
     |  |  |  |
     |  |  |  +-> Ground
     |  |  +----> Clock
     |  +-------> Latch
     +----------> Data

Pinout for SNES gamepads and mice

  +----------------------\
7 | o  o  o  o |  x  x  o  | 1
  +----------------------/
    |  |  |  |          |
    |  |  |  |          +-> Ground
    |  |  |  +------------> Data
    |  |  +---------------> Latch
    |  +------------------> Clock
    +---------------------> Power

Pinout for NES clone (db9) gamepads

      +---------> Clock
      | +-------> Latch
      | | +-----> Data
      | | |
 _____________
5 \ x o o o x / 1
   \ x o x o /
  9 `~~~~~~~' 6
       |   |
       |   +----> Power
       +--------> Ground

Pinout for NES clone (db15) gamepads

    +-----------------> Data
    |             +---> Ground
    |             |
 ___________________
8 \ o x x x x x x o / 1
   \ o x x o x x o /
 15 `~~~~~~~~~~~~~' 9
     |     |     |
     |     |     +----> Clock
     |     +----------> Latch
     +----------------> Power
```

Multisystem joysticks

In the era of 8-bit machines, there was something like de-facto standard for joystick ports. They were all digital, and all used D-Cannon 9 pin connectors (db9). Because of that, a single joystick could be used without hassle on Atari (130, 800XE, 800XL, 2600, 7200), Amiga, Commodore C64, Amstrad CPC, Sinclair ZX Spectrum and many other machines. That's why these joysticks are called "Multisystem".

Now their pinout:

```
      +---------> Right
      | +-------> Left
      | | +-----> Down
```

```
      | | | +---> Up
      | | | |
  _____________
5 \ x o o o o / 1
   \ x o x o /
  9 `~~~~~~~' 6
      |   |
      |   +----> Button
      +--------> Ground
```

However, as time passed, extensions to this standard developed, and these were not compatible with each other:

```
        Atari 130, 800/XL/XE                    MSX


      +---------> Right                        +-----------> Power
      | +-------> Left                         | +---------> Right
      | | +-----> Down                         | | +-------> Left
      | | | +---> Up                           | | | +-----> Down
      | | | |                                  | | | | +---> Up
  _____________                                | | | | |
5 \ x o o o o / 1                          _____________
   \ x o o o /                           5 \ o o o o o / 1
  9 `~~~~~~~' 6                             \ o o o o /
      | | |                                9 `~~~~~~~' 6
      | | +----> Button                        | | | |
      | +------> Power                         | | | +----> Button 1
      +--------> Ground                        | | +------> Button 2
                                               | +--------> Output 3
                                               +----------> Ground

        Amstrad CPC                                Commodore C64


      +---------> Right                        +-----------> Analog Y
      | +-------> Left                         | +---------> Right
      | | +-----> Down                         | | +-------> Left
      | | | +---> Up                           | | | +-----> Down
      | | | |                                  | | | | +---> Up
  _____________                                | | | | |
5 \ x o o o o / 1                          _____________
   \ x o o o /                           5 \ o o o o o / 1
  9 `~~~~~~~' 6                             \ o o o o /
      | | |                                9 `~~~~~~~' 6
      | | +----> Button 1                      | | | |
      | +------> Button 2                      | | | +----> Button
      +--------> Ground                        | | +------> Power
                                               | +--------> Ground
                                               +----------> Analog X

        Sinclair Spectrum +2A/+3                  Amiga 1200


    +-----------> Up                           +-----------> Button 3
    | +---------> Fire                         | +---------> Right
    | |                                        | | +-------> Left
    | |   +-----> Ground                       | | | +-----> Down
    | |   |                                    | | | | +---> Up
    | |   |                                    | | | | |
  _____________                              _____________
5 \ o o x o x / 1                          5 \ o o o o o / 1
   \ o o o o /                               \ o o o o /
  9 `~~~~~~~' 6                             9 `~~~~~~~' 6
     | | | |                                   | | | |
```

```
     | | | +----> Right                  | | | +----> Button 1
     | | +------> Left                   | | +------> Power
     | +--------> Ground                 | +--------> Ground
     +----------> Down                   +----------> Button 2

And there were many others.
```

Multisystem joysticks using db9.c

For the Multisystem joysticks, and their derivatives, the db9.c driver was written. It allows only one joystick / gamepad per parallel port, but the interface is easy to build and works with almost anything.

For the basic 1-button Multisystem joystick you connect its wires to the parallel port like this:

```
(pin  1) -----> Power
(pin 18) -----> Ground

(pin  2) -----> Up
(pin  3) -----> Down
(pin  4) -----> Left
(pin  5) -----> Right
(pin  6) -----> Button 1
```

However, if the joystick is switch based (eg. clicks when you move it), you might or might not, depending on your parallel port, need 10 kOhm pullup resistors on each of the direction and button signals, like this:

```
(pin 2) ------------+------> Up
          Resistor  |
(pin 1) --[10kOhm]--+
```

Try without, and if it doesn't work, add them. For TTL based joysticks / gamepads the pullups are not needed.

For joysticks with two buttons you connect the second button to pin 7 on the parallel port:

```
(pin 7) -----> Button 2
```

And that's it.

On a side note, if you have already built a different adapter for use with the digital joystick driver 0.8.0.2, this is also supported by the db9.c driver, as device type 8. (See section 3.2)

Multisystem joysticks using gamecon.c

For some people just one joystick per parallel port is not enough, and/or want to use them on one parallel port together with NES/SNES/PSX pads. This is possible using the gamecon.c. It supports up to 5 devices of the above types, including 1 and 2 buttons Multisystem joysticks.

However, there is nothing for free. To allow more sticks to be used at once, you need the sticks to be purely switch based (that is non-TTL), and not to need power. Just a plain simple six switches inside. If your joystick can do more (eg. turbofire) you'll need to disable it totally first if you want to use gamecon.c.

Also, the connection is a bit more complex. You'll need a bunch of diodes, and one pullup resistor. First, you connect the Directions and the button the same as for db9, however with the diodes between:

```
            Diodes
(pin 2) -----|<|----> Up
(pin 3) -----|<|----> Down
(pin 4) -----|<|----> Left
```

```
(pin 5) -----|<|----> Right
(pin 6) -----|<|----> Button 1
```

For two button sticks you also connect the other button:

```
(pin 7) -----|<|----> Button 2
```

And finally, you connect the Ground wire of the joystick, like done in this little schematic to Power and Data on the parallel port, as described for the NES / SNES pads in section 2.1 of this file - that is, one data pin for each joystick. The power source is shared:

```
Data    ------------+-----> Ground
          Resistor  |
Power   --[10kOhm]--+
```

And that's all, here we go!

Multisystem joysticks using turbografx.c

The TurboGraFX interface, designed by

> Steffen Schwenke <schwenke@burg-halle.de>

allows up to 7 Multisystem joysticks connected to the parallel port. In Steffen's version, there is support for up to 5 buttons per joystick. However, since this doesn't work reliably on all parallel ports, the turbografx.c driver supports only one button per joystick. For more information on how to build the interface, see:

> http://www2.burg-halle.de/~schwenke/parport.html

Sony Playstation

The PSX controller is supported by the gamecon.c. Pinout of the PSX controller (compatible with DirectPadPro):

```
  +---------+---------+---------+
9 | o  o  o | o  o  o | o  o  o | 1              parallel
   \________|_________|________/                 port pins
    |  |      |  |  |   |
    |  |      |  |  |   +--------->  Clock    ---  (4)
    |  |      |  |  +------------>  Select   ---  (3)
    |  |      |  +---------------->  Power    ---  (5-9)
    |  |      +------------------->  Ground   ---  (18-25)
    |  +-------------------------->  Command  ---  (2)
    +---------------------------->  Data     ---  (one of 10,11,12,13,15)
```

The driver supports these controllers:

- Standard PSX Pad
- NegCon PSX Pad
- Analog PSX Pad (red mode)
- Analog PSX Pad (green mode)
- PSX Rumble Pad
- PSX DDR Pad

Sega

All the Sega controllers are more or less based on the standard 2-button Multisystem joystick. However, since they don't use switches and use TTL logic, the only driver usable with them is the db9.c driver.

Sega Master System

The SMS gamepads are almost exactly the same as normal 2-button Multisystem joysticks. Set the driver to Multi2 mode, use the corresponding parallel port pins, and the following schematic:

```
    +-----------> Power
    | +---------> Right
    | | +-------> Left
    | | | +-----> Down
    | | | | +---> Up
    | | | | |
  _____________
5 \ o o o o o / 1
   \ o o x o /
  9 `~~~~~~~' 6
     | |   |
     | |   +----> Button 1
     | +--------> Ground
     +----------> Button 2
```

Sega Genesis aka MegaDrive

The Sega Genesis (in Europe sold as Sega MegaDrive) pads are an extension to the Sega Master System pads. They use more buttons (3+1, 5+1, 6+1). Use the following schematic:

```
    +-----------> Power
    | +---------> Right
    | | +-------> Left
    | | | +-----> Down
    | | | | +---> Up
    | | | | |
  _____________
5 \ o o o o o / 1
   \ o o o o /
  9 `~~~~~~~' 6
    | | | |
    | | | +----> Button 1
    | | +------> Select
    | +--------> Ground
    +----------> Button 2
```

The Select pin goes to pin 14 on the parallel port:

```
(pin 14) -----> Select
```

The rest is the same as for Multi2 joysticks using db9.c

Sega Saturn

Sega Saturn has eight buttons, and to transfer that, without hacks like Genesis 6 pads use, it needs one more select pin. Anyway, it is still handled by the db9.c driver. Its pinout is very different from anything else. Use this schematic:

```
    +-----------> Select 1
    | +---------> Power
    | | +-------> Up
    | | | +-----> Down
    | | | | +---> Ground
    | | | | |

  _____________
5 \ o o o o o / 1
   \ o o o o /
  9 `~~~~~~~' 6
     | | | |
     | | | +----> Select 2
     | | +------> Right
     | +--------> Left
     +----------> Power
```

Select 1 is pin 14 on the parallel port, Select 2 is pin 16 on the parallel port:

```
(pin 14) -----> Select 1
(pin 16) -----> Select 2
```

The other pins (Up, Down, Right, Left, Power, Ground) are the same as for Multi joysticks using db9.c

Amiga CD32

Amiga CD32 joypad uses the following pinout:

```
    +-----------> Button 3
    | +---------> Right
    | | +-------> Left
    | | | +-----> Down
    | | | | +---> Up
    | | | | |

  _____________
5 \ o o o o o / 1
   \ o o o o /
  9 `~~~~~~~' 6
    | | | |
    | | | +----> Button 1
    | | +------> Power
    | +--------> Ground
    +----------> Button 2
```

It can be connected to the parallel port and driven by db9.c driver. It needs the following wiring:

CD32 pad	Parallel port
1 (Up)	2 (D0)
2 (Down)	3 (D1)
3 (Left)	4 (D2)
4 (Right)	5 (D3)
5 (Button 3)	14 (AUTOFD)
6 (Button 1)	17 (SELIN)
7 (+5V)	1 (STROBE)
8 (Gnd)	18 (Gnd)
9 (Button 2)	7 (D5)

The drivers

There are three drivers for the parallel port interfaces. Each, as described above, allows to connect a different group of joysticks and pads. Here are described their command lines:

gamecon.c

Using gamecon.c you can connect up to five devices to one parallel port. It uses the following kernel/module command line:

```
gamecon.map=port,pad1,pad2,pad3,pad4,pad5
```

Where `port` the number of the parport interface (eg. 0 for parport0).

And pad1 to pad5 are pad types connected to different data input pins (10,11,12,13,15), as described in section 2.1 of this file.

The types are:

Type	Joystick/Pad
0	None
1	SNES pad
2	NES pad
4	Multisystem 1-button joystick
5	Multisystem 2-button joystick
6	N64 pad
7	Sony PSX controller
8	Sony PSX DDR controller
9	SNES mouse

The exact type of the PSX controller type is autoprobed when used, so hot swapping should work (but is not recommended).

Should you want to use more than one of parallel ports at once, you can use gamecon.map2 and gamecon.map3 as additional command line parameters for two more parallel ports.

There are two options specific to PSX driver portion. gamecon.psx_delay sets the command delay when talking to the controllers. The default of 25 should work but you can try lowering it for better performance. If your pads don't respond try raising it until they work. Setting the type to 8 allows the driver to be used with Dance Dance Revolution or similar games. Arrow keys are registered as key presses instead of X and Y axes.

db9.c

Apart from making an interface, there is nothing difficult on using the db9.c driver. It uses the following kernel/module command line:

```
db9.dev=port,type
```

Where `port` is the number of the parport interface (eg. 0 for parport0).

Caveat here: This driver only works on bidirectional parallel ports. If your parallel port is recent enough, you should have no trouble with this. Old parallel ports may not have this feature.

Type is the type of joystick or pad attached:

Type	Joystick/Pad
0	None
1	Multisystem 1-button joystick
2	Multisystem 2-button joystick
3	Genesis pad (3+1 buttons)
5	Genesis pad (5+1 buttons)
6	Genesis pad (6+2 buttons)
7	Saturn pad (8 buttons)
8	Multisystem 1-button joystick (v0.8.0.2 pin-out)
9	Two Multisystem 1-button joysticks (v0.8.0.2 pin-out)
10	Amiga CD32 pad

Should you want to use more than one of these joysticks/pads at once, you can use db9.dev2 and db9.dev3 as additional command line parameters for two more joysticks/pads.

turbografx.c

The turbografx.c driver uses a very simple kernel/module command line:

```
turbografx.map=port,js1,js2,js3,js4,js5,js6,js7
```

Where `port` is the number of the parport interface (eg. 0 for parport0).

`jsX` is the number of buttons the Multisystem joysticks connected to the interface ports 1-7 have. For a standard multisystem joystick, this is 1.

Should you want to use more than one of these interfaces at once, you can use turbografx.map2 and turbografx.map3 as additional command line parameters for two more interfaces.

PC parallel port pinout

```
                .-----------------------------------------.
At the PC:      \ 13 12 11 10  9  8  7  6  5  4  3  2  1 /
                 \  25 24 23 22 21 20 19 18 17 16 15 14 /
                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Pin	Name	Description
1	/STROBE	Strobe
2-9	D0-D7	Data Bit 0-7
10	/ACK	Acknowledge
11	BUSY	Busy
12	PE	Paper End
13	SELIN	Select In
14	/AUTOFD	Autofeed
15	/ERROR	Error
16	/INIT	Initialize
17	/SEL	Select
18-25	GND	Signal Ground

That's all, folks! Have fun!

N-Trig touchscreen Driver

This driver provides support for N-Trig pen and multi-touch sensors. Single and multi-touch events are translated to the appropriate protocols for the hid and input systems. Pen events are sufficiently hid compliant and are left to the hid core. The driver also provides additional filtering and utility functions accessible with sysfs and module parameters.

This driver has been reported to work properly with multiple N-Trig devices attached.

Parameters

Note: values set at load time are global and will apply to all applicable devices. Adjusting parameters with sysfs will override the load time values, but only for that one device.

The following parameters are used to configure filters to reduce noise:

activate_slack	number of fingers to ignore before processing events
activation_height, activation_width	size threshold to activate immediately
min_height, min_width	size threshold bellow which fingers are ignored both to decide activation and during activity
deactivate_slack	the number of “no contact” frames to ignore before propagating the end of activity events

When the last finger is removed from the device, it sends a number of empty frames. By holding off on deactivation for a few frames we can tolerate false erroneous disconnects, where the sensor may mistakenly not detect a finger that is still present. Thus deactivate_slack addresses problems where a users might see breaks in lines during drawing, or drop an object during a long drag.

Additional sysfs items

These nodes just provide easy access to the ranges reported by the device.

sensor_logical_height, sensor_logical_width	the range for positions reported during activity
sensor_physical_height, sensor_physical_width	internal ranges not used for normal events but useful for tuning

All N-Trig devices with product id of 1 report events in the ranges of

- X: 0-9600
- Y: 0-7200

However not all of these devices have the same physical dimensions. Most seem to be 12” sensors (Dell Latitude XT and XT2 and the HP TX2), and at least one model (Dell Studio 17) has a 17” sensor. The ratio of physical to logical sizes is used to adjust the size based filter parameters.

Filtering

With the release of the early multi-touch firmwares it became increasingly obvious that these sensors were prone to erroneous events. Users reported seeing both inappropriately dropped contact and ghosts, contacts reported where no finger was actually touching the screen.

Deactivation slack helps prevent dropped contact for single touch use, but does not address the problem of dropping one of more contacts while other contacts are still active. Drops in the multi-touch context require additional processing and should be handled in tandem with tacking.

As observed ghost contacts are similar to actual use of the sensor, but they seem to have different profiles. Ghost activity typically shows up as small short lived touches. As such, I assume that the longer the continuous stream of events the more likely those events are from a real contact, and that the larger the size of each contact the more likely it is real. Balancing the goals of preventing ghosts and accepting real events quickly (to minimize user observable latency), the filter accumulates confidence for incoming

events until it hits thresholds and begins propagating. In the interest in minimizing stored state as well as the cost of operations to make a decision, I've kept that decision simple.

Time is measured in terms of the number of fingers reported, not frames since the probability of multiple simultaneous ghosts is expected to drop off dramatically with increasing numbers. Rather than accumulate weight as a function of size, I just use it as a binary threshold. A sufficiently large contact immediately overrides the waiting period and leads to activation.

Setting the activation size thresholds to large values will result in deciding primarily on activation slack. If you see longer lived ghosts, turning up the activation slack while reducing the size thresholds may suffice to eliminate the ghosts while keeping the screen quite responsive to firm taps.

Contacts continue to be filtered with min_height and min_width even after the initial activation filter is satisfied. The intent is to provide a mechanism for filtering out ghosts in the form of an extra finger while you actually are using the screen. In practice this sort of ghost has been far less problematic or relatively rare and I've left the defaults set to 0 for both parameters, effectively turning off that filter.

I don't know what the optimal values are for these filters. If the defaults don't work for you, please play with the parameters. If you do find other values more comfortable, I would appreciate feedback.

The calibration of these devices does drift over time. If ghosts or contact dropping worsen and interfere with the normal usage of your device, try recalibrating it.

Calibration

The N-Trig windows tools provide calibration and testing routines. Also an unofficial unsupported set of user space tools including a calibrator is available at: http://code.launchpad.net/~rafi-seas/+junk/ntrig_calib

Tracking

As of yet, all tested N-Trig firmwares do not track fingers. When multiple contacts are active they seem to be sorted primarily by Y position.

rotary-encoder - a generic driver for GPIO connected devices

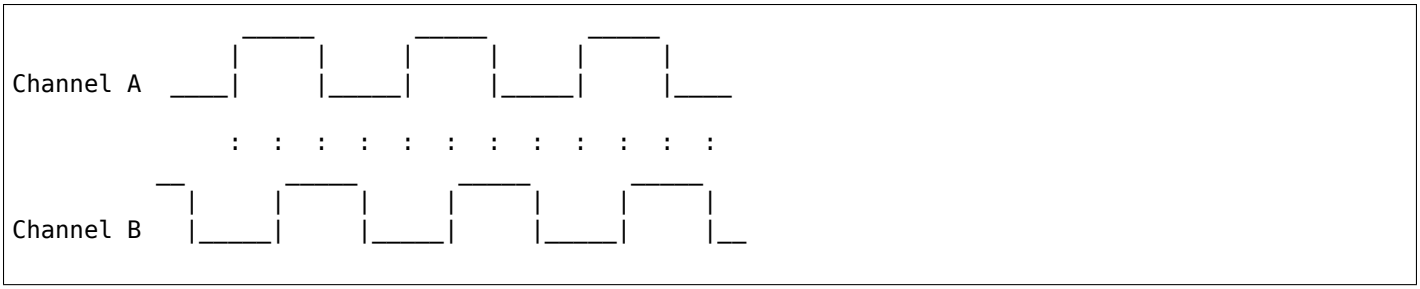
Author Daniel Mack <daniel@caiaq.de>, Feb 2009

Function

Rotary encoders are devices which are connected to the CPU or other peripherals with two wires. The outputs are phase-shifted by 90 degrees and by triggering on falling and rising edges, the turn direction can be determined.

Some encoders have both outputs low in stable states, others also have a stable state with both outputs high (half-period mode) and some have a stable state in all steps (quarter-period mode).

The phase diagram of these two outputs look like this:



```
              :  :  :  :  :  :  :  :  :  :  :  :
Event         a  b  c  d  a  b  c  d  a  b  c  d

              |<-------->|
                one step

              |<-->|
                one step (half-period mode)

              |<>|
                one step (quarter-period mode)
```

For more information, please see https://en.wikipedia.org/wiki/Rotary_encoder

Events / state machine

In half-period mode, state a) and c) above are used to determine the rotational direction based on the last stable state. Events are reported in states b) and d) given that the new stable state is different from the last (i.e. the rotation was not reversed half-way).

Otherwise, the following apply:

1. **Rising edge on channel A, channel B in low state** This state is used to recognize a clockwise turn
2. **Rising edge on channel B, channel A in high state** When entering this state, the encoder is put into 'armed' state, meaning that there it has seen half the way of a one-step transition.
3. **Falling edge on channel A, channel B in high state** This state is used to recognize a counter-clockwise turn
4. **Falling edge on channel B, channel A in low state** Parking position. If the encoder enters this state, a full transition should have happened, unless it flipped back on half the way. The 'armed' state tells us about that.

Platform requirements

As there is no hardware dependent call in this driver, the platform it is used with must support gpiolib. Another requirement is that IRQs must be able to fire on both edges.

Board integration

To use this driver in your system, register a platform_device with the name 'rotary-encoder' and associate the IRQs and some specific platform data with it. Because the driver uses generic device properties, this can be done either via device tree, ACPI, or using static board files, like in example below:

```
/* board support file example */

#include <linux/input.h>
#include <linux/gpio/machine.h>
#include <linux/property.h>

#define GPIO_ROTARY_A 1
#define GPIO_ROTARY_B 2

static struct gpiod_lookup_table rotary_encoder_gpios = {
        .dev_id = "rotary-encoder.0",
        .table = {
                GPIO_LOOKUP_IDX("gpio-0",
```

```
                                GPIO_ROTARY_A, NULL, 0, GPIO_ACTIVE_LOW),
                GPIO_LOOKUP_IDX("gpio-0",
                                GPIO_ROTARY_B, NULL, 1, GPIO_ACTIVE_HIGH),
                { },
        },
};

static const struct property_entry rotary_encoder_properties[] __initconst = {
        PROPERTY_ENTRY_INTEGER("rotary-encoder,steps-per-period", u32, 24),
        PROPERTY_ENTRY_INTEGER("linux,axis",                      u32, ABS_X),
        PROPERTY_ENTRY_INTEGER("rotary-encoder,relative_axis",    u32, 0),
        { },
};

static struct platform_device rotary_encoder_device = {
        .name           = "rotary-encoder",
        .id             = 0,
};

...

gpiod_add_lookup_table(&rotary_encoder_gpios);
device_add_properties(&rotary_encoder_device, rotary_encoder_properties);
platform_device_register(&rotary_encoder_device);

...
```

Please consult device tree binding documentation to see all properties supported by the driver.

Sentelic Touchpad

Copyright © 2002-2011 Sentelic Corporation.

Last update Dec-07-2011

Finger Sensing Pad Intellimouse Mode (scrolling wheel, 4th and 5th buttons)

1. MSID 4: Scrolling wheel mode plus Forward page(4th button) and Backward page (5th button)

1. Set sample rate to 200;
2. Set sample rate to 200;
3. Set sample rate to 80;
4. Issuing the “Get device ID” command (0xF2) and waits for the response;
5. FSP will respond 0x04.

```
Packet 1
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |Y|X|y|x|1|M|R|L|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 | | |B|F|W|W|W|W|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7 => Y overflow
        Bit6 => X overflow
        Bit5 => Y sign bit
        Bit4 => X sign bit
        Bit3 => 1
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
```

```
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: X Movement(9-bit 2's complement integers)
Byte 3: Y Movement(9-bit 2's complement integers)
Byte 4: Bit3~Bit0 => the scrolling wheel's movement since the last data report.
                     valid values, -8 ~ +7
        Bit4 => 1 = 4th mouse button is pressed, Forward one page.
                0 = 4th mouse button is not pressed.
        Bit5 => 1 = 5th mouse button is pressed, Backward one page.
                0 = 5th mouse button is not pressed.
```

2. MSID 6: Horizontal and Vertical scrolling

- Set bit 1 in register 0x40 to 1

FSP replaces scrolling wheel's movement as 4 bits to show horizontal and vertical scrolling.

```
Packet 1
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |Y|X|y|x|1|M|R|L|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 | | |B|F|r|l|u|d|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7 => Y overflow
        Bit6 => X overflow
        Bit5 => Y sign bit
        Bit4 => X sign bit
        Bit3 => 1
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: X Movement(9-bit 2's complement integers)
Byte 3: Y Movement(9-bit 2's complement integers)
Byte 4: Bit0 => the Vertical scrolling movement downward.
        Bit1 => the Vertical scrolling movement upward.
        Bit2 => the Horizontal scrolling movement leftward.
        Bit3 => the Horizontal scrolling movement rightward.
        Bit4 => 1 = 4th mouse button is pressed, Forward one page.
                0 = 4th mouse button is not pressed.
        Bit5 => 1 = 5th mouse button is pressed, Backward one page.
                0 = 5th mouse button is not pressed.
```

3. MSID 7

FSP uses 2 packets (8 Bytes) to represent Absolute Position. so we have PACKET NUMBER to identify packets.

> If PACKET NUMBER is 0, the packet is Packet 1. If PACKET NUMBER is 1, the packet is Packet 2. Please count this number in program.

MSID6 special packet will be enable at the same time when enable MSID 7.

Absolute position for STL3886-G0

1. Set bit 2 or 3 in register 0x40 to 1
2. Set bit 6 in register 0x40 to 1

```
Packet 1 (ABSOLUTE POSITION)
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |0|1|V|1|1|M|R|L|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 |r|l|d|u|X|X|Y|Y|
      |---------------|     |---------------|    |---------------|    |---------------|
```

```
Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordination packet
                  => 10, Notify packet
        Bit5 => valid bit
        Bit4 => 1
        Bit3 => 1
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: X coordinate (xpos[9:2])
Byte 3: Y coordinate (ypos[9:2])
Byte 4: Bit1~Bit0 => Y coordinate (xpos[1:0])
        Bit3~Bit2 => X coordinate (ypos[1:0])
        Bit4 => scroll up
        Bit5 => scroll down
        Bit6 => scroll left
        Bit7 => scroll right

Notify Packet for G0
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |1|0|0|1|1|M|R|L|  2  |C|C|C|C|C|C|C|C|  3 |M|M|M|M|M|M|M|M|  4 |0|0|0|0|0|0|0|0|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordination packet
                  => 10, Notify packet
        Bit5 => 0
        Bit4 => 1
        Bit3 => 1
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: Message Type => 0x5A (Enable/Disable status packet)
        Mode Type => 0xA5 (Normal/Icon mode status)
Byte 3: Message Type => 0x00 (Disabled)
                     => 0x01 (Enabled)
        Mode Type    => 0x00 (Normal)
                     => 0x01 (Icon)
Byte 4: Bit7~Bit0 => Don't Care
```

Absolute position for STL3888-Ax

```
Packet 1 (ABSOLUTE POSITION)
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |0|1|V|A|1|L|0|1|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 |x|x|y|y|X|X|Y|Y|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordination packet
                  => 10, Notify packet
                  => 11, Normal data packet with on-pad click
        Bit5 => Valid bit, 0 means that the coordinate is invalid or finger up.
                When both fingers are up, the last two reports have zero valid
                bit.
        Bit4 => arc
        Bit3 => 1
        Bit2 => Left Button, 1 is pressed, 0 is released.
        Bit1 => 0
```

```
        Bit0 => 1
Byte 2: X coordinate (xpos[9:2])
Byte 3: Y coordinate (ypos[9:2])
Byte 4: Bit1~Bit0 => Y coordinate (xpos[1:0])
        Bit3~Bit2 => X coordinate (ypos[1:0])
        Bit5~Bit4 => y1_g
        Bit7~Bit6 => x1_g

Packet 2 (ABSOLUTE POSITION)
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |0|1|V|A|1|R|1|0|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 |x|x|y|y|X|X|Y|Y|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordinates packet
                  => 10, Notify packet
                  => 11, Normal data packet with on-pad click
        Bit5 => Valid bit, 0 means that the coordinate is invalid or finger up.
                When both fingers are up, the last two reports have zero valid
                bit.
        Bit4 => arc
        Bit3 => 1
        Bit2 => Right Button, 1 is pressed, 0 is released.
        Bit1 => 1
        Bit0 => 0
Byte 2: X coordinate (xpos[9:2])
Byte 3: Y coordinate (ypos[9:2])
Byte 4: Bit1~Bit0 => Y coordinate (xpos[1:0])
        Bit3~Bit2 => X coordinate (ypos[1:0])
        Bit5~Bit4 => y2_g
        Bit7~Bit6 => x2_g

Notify Packet for STL3888-Ax
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |1|0|1|P|1|M|R|L|  2  |C|C|C|C|C|C|C|C|  3 |0|0|F|F|0|0|0|i|  4 |r|l|d|u|0|0|0|0|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordinates packet
                  => 10, Notify packet
                  => 11, Normal data packet with on-pad click
        Bit5 => 1
        Bit4 => when in absolute coordinates mode (valid when EN_PKT_GO is 1):
                0: left button is generated by the on-pad command
                1: left button is generated by the external button
        Bit3 => 1
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: Message Type => 0xB7 (Multi Finger, Multi Coordinate mode)
Byte 3: Bit7~Bit6 => Don't care
        Bit5~Bit4 => Number of fingers
        Bit3~Bit1 => Reserved
        Bit0 => 1: enter gesture mode; 0: leaving gesture mode
Byte 4: Bit7 => scroll right button
        Bit6 => scroll left button
        Bit5 => scroll down button
        Bit4 => scroll up button
            * Note that if gesture and additional button (Bit4~Bit7)
            happen at the same time, the button information will not
            be sent.
```

```
        Bit3~Bit0 => Reserved
```

Sample sequence of Multi-finger, Multi-coordinate mode:

notify packet (valid bit == 1), abs pkt 1, abs pkt 2, abs pkt 1, abs pkt 2, ..., notify packet (valid bit == 0)

Absolute position for STL3888-B0

```
Packet 1(ABSOLUTE POSITION)
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |0|1|V|F|1|0|R|L|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 |r|l|u|d|X|X|Y|Y|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordinates packet
                  => 10, Notify packet
                  => 11, Normal data packet with on-pad click
        Bit5 => Valid bit, 0 means that the coordinate is invalid or finger up.
                When both fingers are up, the last two reports have zero valid
                bit.
        Bit4 => finger up/down information. 1: finger down, 0: finger up.
        Bit3 => 1
        Bit2 => finger index, 0 is the first finger, 1 is the second finger.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: X coordinate (xpos[9:2])
Byte 3: Y coordinate (ypos[9:2])
Byte 4: Bit1~Bit0 => Y coordinate (xpos[1:0])
        Bit3~Bit2 => X coordinate (ypos[1:0])
        Bit4 => scroll down button
        Bit5 => scroll up button
        Bit6 => scroll left button
        Bit7 => scroll right button

Packet 2 (ABSOLUTE POSITION)
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |0|1|V|F|1|1|R|L|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 |r|l|u|d|X|X|Y|Y|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordination packet
                  => 10, Notify packet
                  => 11, Normal data packet with on-pad click
        Bit5 => Valid bit, 0 means that the coordinate is invalid or finger up.
                When both fingers are up, the last two reports have zero valid
                bit.
        Bit4 => finger up/down information. 1: finger down, 0: finger up.
        Bit3 => 1
        Bit2 => finger index, 0 is the first finger, 1 is the second finger.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: X coordinate (xpos[9:2])
Byte 3: Y coordinate (ypos[9:2])
Byte 4: Bit1~Bit0 => Y coordinate (xpos[1:0])
        Bit3~Bit2 => X coordinate (ypos[1:0])
        Bit4 => scroll down button
        Bit5 => scroll up button
        Bit6 => scroll left button
```

```
        Bit7 => scroll right button
```

Notify Packet for STL3888-B0:

```
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |1|0|1|P|1|M|R|L|  2  |C|C|C|C|C|C|C|C|  3 |0|0|F|F|0|0|0|i|  4 |r|l|u|d|0|0|0|0|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordination packet
                  => 10, Notify packet
                  => 11, Normal data packet with on-pad click
        Bit5 => 1
        Bit4 => when in absolute coordinates mode (valid when EN_PKT_GO is 1):
                0: left button is generated by the on-pad command
                1: left button is generated by the external button
        Bit3 => 1
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: Message Type => 0xB7 (Multi Finger, Multi Coordinate mode)
Byte 3: Bit7~Bit6 => Don't care
        Bit5~Bit4 => Number of fingers
        Bit3~Bit1 => Reserved
        Bit0 => 1: enter gesture mode; 0: leaving gesture mode
Byte 4: Bit7 => scroll right button
        Bit6 => scroll left button
        Bit5 => scroll up button
        Bit4 => scroll down button
            * Note that if gesture and additional button(Bit4~Bit7)
            happen at the same time, the button information will not
            be sent.
        Bit3~Bit0 => Reserved
```

Sample sequence of Multi-finger, Multi-coordinate mode:

> notify packet (valid bit == 1), abs pkt 1, abs pkt 2, abs pkt 1, abs pkt 2, ..., notify packet (valid bit == 0)

Absolute position for STL3888-Cx and STL3888-Dx

```
Single Finger, Absolute Coordinate Mode (SFAC)
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |0|1|0|P|1|M|R|L|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 |r|l|B|F|X|X|Y|Y|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordinates packet
                  => 10, Notify packet
        Bit5 => Coordinate mode(always 0 in SFAC mode):
                0: single-finger absolute coordinates (SFAC) mode
                1: multi-finger, multiple coordinates (MFMC) mode
        Bit4 => 0: The LEFT button is generated by on-pad command (OPC)
                1: The LEFT button is generated by external button
                Default is 1 even if the LEFT button is not pressed.
        Bit3 => Always 1, as specified by PS/2 protocol.
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
```

```
Byte 2: X coordinate (xpos[9:2])
Byte 3: Y coordinate (ypos[9:2])
Byte 4: Bit1~Bit0 => Y coordinate (xpos[1:0])
        Bit3~Bit2 => X coordinate (ypos[1:0])
        Bit4 => 4th mouse button(forward one page)
        Bit5 => 5th mouse button(backward one page)
        Bit6 => scroll left button
        Bit7 => scroll right button

Multi Finger, Multiple Coordinates Mode (MFMC):
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |0|1|1|P|1|F|R|L|  2  |X|X|X|X|X|X|X|X|  3 |Y|Y|Y|Y|Y|Y|Y|Y|  4 |r|l|B|F|X|X|Y|Y|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordination packet
                  => 10, Notify packet
        Bit5 => Coordinate mode (always 1 in MFMC mode):
                  0: single-finger absolute coordinates (SFAC) mode
                  1: multi-finger, multiple coordinates (MFMC) mode
        Bit4 => 0: The LEFT button is generated by on-pad command (OPC)
                  1: The LEFT button is generated by external button
                  Default is 1 even if the LEFT button is not pressed.
        Bit3 => Always 1, as specified by PS/2 protocol.
        Bit2 => Finger index, 0 is the first finger, 1 is the second finger.
                  If bit 1 and 0 are all 1 and bit 4 is 0, the middle external
                  button is pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: X coordinate (xpos[9:2])
Byte 3: Y coordinate (ypos[9:2])
Byte 4: Bit1~Bit0 => Y coordinate (xpos[1:0])
        Bit3~Bit2 => X coordinate (ypos[1:0])
        Bit4 => 4th mouse button(forward one page)
        Bit5 => 5th mouse button(backward one page)
        Bit6 => scroll left button
        Bit7 => scroll right button
```

When one of the two fingers is up, the device will output four consecutive MFMC#0 report packets with zero X and Y to represent 1st finger is up or four consecutive MFMC#1 report packets with zero X and Y to represent that the 2nd finger is up. On the other hand, if both fingers are up, the device will output four consecutive single-finger, absolute coordinate(SFAC) packets with zero X and Y.

Notify Packet for STL3888-Cx/Dx:

```
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |1|0|0|P|1|M|R|L|  2  |C|C|C|C|C|C|C|C|  3 |0|0|F|F|0|0|0|i|  4 |r|l|u|d|0|0|0|0|
      |---------------|     |---------------|    |---------------|    |---------------|

Byte 1: Bit7~Bit6 => 00, Normal data packet
                  => 01, Absolute coordinates packet
                  => 10, Notify packet
        Bit5 => Always 0
        Bit4 => 0: The LEFT button is generated by on-pad command(OPC)
                  1: The LEFT button is generated by external button
                  Default is 1 even if the LEFT button is not pressed.
        Bit3 => 1
        Bit2 => Middle Button, 1 is pressed, 0 is not pressed.
        Bit1 => Right Button, 1 is pressed, 0 is not pressed.
        Bit0 => Left Button, 1 is pressed, 0 is not pressed.
Byte 2: Message type:
```

```
        0xba => gesture information
        0xc0 => one finger hold-rotating gesture
Byte 3: The first parameter for the received message:
        0xba => gesture ID (refer to the 'Gesture ID' section)
        0xc0 => region ID
Byte 4: The second parameter for the received message:
        0xba => N/A
        0xc0 => finger up/down information
```

Sample sequence of Multi-finger, Multi-coordinates mode:

notify packet (valid bit == 1), MFMC packet 1 (byte 1, bit 2 == 0), MFMC packet 2 (byte 1, bit 2 == 1), MFMC packet 1, MFMC packet 2, ..., notify packet (valid bit == 0)

That is, when the device is in MFMC mode, the host will receive interleaved absolute coordinate packets for each finger.

FSP Enable/Disable packet

```
Bit 7 6 5 4 3 2 1 0       7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0      7 6 5 4 3 2 1 0
BYTE  |---------------|BYTE |---------------|BYTE|---------------|BYTE|---------------|
  1   |Y|X|0|0|1|M|R|L|  2  |0|1|0|1|1|0|1|E|  3 | | | | | | | |  4 | | | | | | | |
      |---------------|     |---------------|    |---------------|    |---------------|

FSP will send out enable/disable packet when FSP receive PS/2 enable/disable
command. Host will receive the packet which Middle, Right, Left button will
be set. The packet only use byte 0 and byte 1 as a pattern of original packet.
Ignore the other bytes of the packet.

Byte 1: Bit7 => 0, Y overflow
        Bit6 => 0, X overflow
        Bit5 => 0, Y sign bit
        Bit4 => 0, X sign bit
        Bit3 => 1
        Bit2 => 1, Middle Button
        Bit1 => 1, Right Button
        Bit0 => 1, Left Button
Byte 2: Bit7~1 => (0101101b)
        Bit0 => 1 = Enable
                0 = Disable
Byte 3: Don't care
Byte 4: Don't care (MOUSE ID 3, 4)
Byte 5~8: Don't care (Absolute packet)
```

PS/2 Command Set

FSP supports basic PS/2 commanding set and modes, refer to following URL for details about PS/2 commands:

http://www.computer-engineering.org/ps2mouse/

Programming Sequence for Determining Packet Parsing Flow

1. Identify FSP by reading device ID(0x00) and version(0x01) register
2. For FSP version < STL3888 Cx, determine number of buttons by reading the ‘test mode status’ (0x20) register:

```
buttons = reg[0x20] & 0x30

if buttons == 0x30 or buttons == 0x20:
        # two/four buttons
        Refer to 'Finger Sensing Pad PS/2 Mouse Intellimouse'
        section A for packet parsing detail(ignore byte 4, bit ~ 7)
elif buttons == 0x10:
        # 6 buttons
        Refer to 'Finger Sensing Pad PS/2 Mouse Intellimouse'
        section B for packet parsing detail
elif buttons == 0x00:
        # 6 buttons
        Refer to 'Finger Sensing Pad PS/2 Mouse Intellimouse'
        section A for packet parsing detail
```

3. **For FSP version >= STL3888 Cx:** Refer to 'Finger Sensing Pad PS/2 Mouse Intellimouse' section A for packet parsing detail (ignore byte 4, bit ~ 7)

Programming Sequence for Register Reading/Writing

Register inversion requirement:

Following values needed to be inverted(the '~' operator in C) before being sent to FSP:

```
0xe8, 0xe9, 0xee, 0xf2, 0xf3 and 0xff.
```

Register swapping requirement:

Following values needed to have their higher 4 bits and lower 4 bits being swapped before being sent to FSP:

```
10, 20, 40, 60, 80, 100 and 200.
```

Register reading sequence:

1. send 0xf3 PS/2 command to FSP;
2. send 0x66 PS/2 command to FSP;
3. send 0x88 PS/2 command to FSP;
4. send 0xf3 PS/2 command to FSP;

5. if the register address being to read is not required to be inverted(refer to the 'Register inversion requirement' section), goto step 6

1. send 0x68 PS/2 command to FSP;
2. send the inverted register address to FSP and goto step 8;

6. if the register address being to read is not required to be swapped(refer to the 'Register swapping requirement' section), goto step 7

1. send 0xcc PS/2 command to FSP;
2. send the swapped register address to FSP and goto step 8;
7. send 0x66 PS/2 command to FSP;
1. send the original register address to FSP and goto step 8;
8. send 0xe9(status request) PS/2 command to FSP;

9. the 4th byte of the response read from FSP should be the requested register value(?? indicates don't care byte):

```
host: 0xe9
3888: 0xfa (??) (??) (val)
```

- Note that since the Cx release, the hardware will return 1's complement of the register value at the 3rd byte of status request result:

```
host: 0xe9
3888: 0xfa (??) (~val) (val)
```

Register writing sequence:

1. send 0xf3 PS/2 command to FSP;

2. if the register address being to write is not required to be inverted(refer to the 'Register inversion requirement' section), goto step 3

   1. send 0x74 PS/2 command to FSP;
   2. send the inverted register address to FSP and goto step 5;

3. if the register address being to write is not required to be swapped(refer to the 'Register swapping requirement' section), goto step 4

   1. send 0x77 PS/2 command to FSP;
   2. send the swapped register address to FSP and goto step 5;

4. send 0x55 PS/2 command to FSP;

   1. send the register address to FSP and goto step 5;

5. send 0xf3 PS/2 command to FSP;

6. if the register value being to write is not required to be inverted(refer to the 'Register inversion requirement' section), goto step 7

   1. send 0x47 PS/2 command to FSP;
   2. send the inverted register value to FSP and goto step 9;

7. if the register value being to write is not required to be swapped(refer to the 'Register swapping requirement' section), goto step 8

   1. send 0x44 PS/2 command to FSP;
   2. send the swapped register value to FSP and goto step 9;

8. send 0x33 PS/2 command to FSP;

   1. send the register value to FSP;

9. the register writing sequence is completed.

- Since the Cx release, the hardware will return 1's complement of the register value at the 3rd byte of status request result. Host can optionally send another 0xe9 (status request) PS/2 command to FSP at the end of register writing to verify that the register writing operation is successful (?? indicates don't care byte):

```
host: 0xe9
3888: 0xfa (??) (~val) (val)
```

Programming Sequence for Page Register Reading/Writing

In order to overcome the limitation of maximum number of registers supported, the hardware separates register into different groups called 'pages.' Each page is able to include up to 255 registers.

The default page after power up is 0x82; therefore, if one has to get access to register 0x8301, one has to use following sequence to switch to page 0x83, then start reading/writing from/to offset 0x01 by using the register read/write sequence described in previous section.

Page register reading sequence:

1. send 0xf3 PS/2 command to FSP;
2. send 0x66 PS/2 command to FSP;
3. send 0x88 PS/2 command to FSP;
4. send 0xf3 PS/2 command to FSP;
5. send 0x83 PS/2 command to FSP;
6. send 0x88 PS/2 command to FSP;
7. send 0xe9(status request) PS/2 command to FSP;
8. the response read from FSP should be the requested page value.

Page register writing sequence:

1. send 0xf3 PS/2 command to FSP;
2. send 0x38 PS/2 command to FSP;
3. send 0x88 PS/2 command to FSP;
4. send 0xf3 PS/2 command to FSP;
5. if the page address being written is not required to be inverted(refer to the 'Register inversion requirement' section), goto step 6
    1. send 0x47 PS/2 command to FSP;
    2. send the inverted page address to FSP and goto step 9;
6. if the page address being written is not required to be swapped(refer to the 'Register swapping requirement' section), goto step 7
    1. send 0x44 PS/2 command to FSP;
    2. send the swapped page address to FSP and goto step 9;
7. send 0x33 PS/2 command to FSP;
8. send the page address to FSP;
9. the page register writing sequence is completed.

Gesture ID

Unlike other devices which sends multiple fingers' coordinates to host, FSP processes multiple fingers' coordinates internally and convert them into a 8 bits integer, namely 'Gesture ID.' Following is a list of supported gesture IDs:

ID	Description
0x86	2 finger straight up
0x82	2 finger straight down
0x80	2 finger straight right
0x84	2 finger straight left
0x8f	2 finger zoom in
0x8b	2 finger zoom out
0xc0	2 finger curve, counter clockwise
0xc4	2 finger curve, clockwise
0x2e	3 finger straight up
0x2a	3 finger straight down
0x28	3 finger straight right
0x2c	3 finger straight left
0x38	palm

Register Listing

Registers are represented in 16 bits values. The higher 8 bits represent the page address and the lower 8 bits represent the relative offset within that particular page. Refer to the 'Programming Sequence for Page Register Reading/Writing' section for instructions on how to change current page address:

```
offset width           default r/w     name
0x8200 bit7~bit0       0x01    RO      device ID

0x8201 bit7~bit0               RW      version ID
                                       0xc1: STL3888 Ax
                                       0xd0 ~ 0xd2: STL3888 Bx
                                       0xe0 ~ 0xe1: STL3888 Cx
                                       0xe2 ~ 0xe3: STL3888 Dx

0x8202 bit7~bit0       0x01    RO      vendor ID

0x8203 bit7~bit0       0x01    RO      product ID

0x8204 bit3~bit0       0x01    RW      revision ID

0x820b                                 test mode status 1
       bit3            1       RO      0: rotate 180 degree
                                       1: no rotation
                                       *only supported by H/W prior to Cx

0x820f                                 register file page control
       bit2            0       RW      1: rotate 180 degree
                                       0: no rotation
                                       *supported since Cx

       bit0            0       RW      1 to enable page 1 register files
                                       *only supported by H/W prior to Cx

0x8210                         RW      system control 1
       bit0            1       RW      Reserved, must be 1
       bit1            0       RW      Reserved, must be 0
       bit4            0       RW      Reserved, must be 0
       bit5            1       RW      register clock gating enable
                                       0: read only, 1: read/write enable
       (Note that following registers does not require clock gating being
       enabled prior to write: 05 06 07 08 09 0c 0f 10 11 12 16 17 18 23 2e
       40 41 42 43.  In addition to that, this bit must be 1 when gesture
       mode is enabled)
```

```
0x8220                                  test mode status
       bit5~bit4               RO       number of buttons
                                        11 => 2, lbtn/rbtn
                                        10 => 4, lbtn/rbtn/scru/scrd
                                        01 => 6, lbtn/rbtn/scru/scrd/scrl/scrr
                                        00 => 6, lbtn/rbtn/scru/scrd/fbtn/bbtn
                                        *only supported by H/W prior to Cx

0x8231                          RW      on-pad command detection
       bit7            0        RW      on-pad command left button down tag
                                        enable
                                        0: disable, 1: enable
                                        *only supported by H/W prior to Cx

0x8234                          RW      on-pad command control 5
       bit4~bit0       0x05     RW      XLO in 0s/4/1, so 03h = 0010.1b = 2.5
       (Note that position unit is in 0.5 scanline)
                                        *only supported by H/W prior to Cx

       bit7            0        RW      on-pad tap zone enable
                                        0: disable, 1: enable
                                        *only supported by H/W prior to Cx

0x8235                          RW      on-pad command control 6
       bit4~bit0       0x1d     RW      XHI in 0s/4/1, so 19h = 1100.1b = 12.5
       (Note that position unit is in 0.5 scanline)
                                        *only supported by H/W prior to Cx

0x8236                          RW      on-pad command control 7
       bit4~bit0       0x04     RW      YLO in 0s/4/1, so 03h = 0010.1b = 2.5
       (Note that position unit is in 0.5 scanline)
                                        *only supported by H/W prior to Cx

0x8237                          RW      on-pad command control 8
       bit4~bit0       0x13     RW      YHI in 0s/4/1, so 11h = 1000.1b = 8.5
       (Note that position unit is in 0.5 scanline)
                                        *only supported by H/W prior to Cx

0x8240                          RW      system control 5
       bit1            0        RW      FSP Intellimouse mode enable
                                        0: disable, 1: enable
                                        *only supported by H/W prior to Cx

       bit2            0        RW      movement + abs. coordinate mode enable
                                        0: disable, 1: enable
       (Note that this function has the functionality of bit 1 even when
       bit 1 is not set. However, the format is different from that of bit 1.
       In addition, when bit 1 and bit 2 are set at the same time, bit 2 will
       override bit 1.)
                                        *only supported by H/W prior to Cx

       bit3            0        RW      abs. coordinate only mode enable
                                        0: disable, 1: enable
       (Note that this function has the functionality of bit 1 even when
       bit 1 is not set. However, the format is different from that of bit 1.
       In addition, when bit 1, bit 2 and bit 3 are set at the same time,
       bit 3 will override bit 1 and 2.)
                                        *only supported by H/W prior to Cx

       bit5            0        RW      auto switch enable
                                        0: disable, 1: enable
                                        *only supported by H/W prior to Cx
```

```
       bit6            0       RW      G0 abs. + notify packet format enable
                                       0: disable, 1: enable
       (Note that the absolute/relative coordinate output still depends on
       bit 2 and 3.  That is, if any of those bit is 1, host will receive
       absolute coordinates; otherwise, host only receives packets with
       relative coordinate.)
                                       *only supported by H/W prior to Cx

       bit7            0       RW      EN_PS2_F2: PS/2 gesture mode 2nd
                                       finger packet enable
                                       0: disable, 1: enable
                                       *only supported by H/W prior to Cx

0x8243                         RW      on-pad control
       bit0            0       RW      on-pad control enable
                                       0: disable, 1: enable
       (Note that if this bit is cleared, bit 3/5 will be ineffective)
                                       *only supported by H/W prior to Cx

       bit3            0       RW      on-pad fix vertical scrolling enable
                                       0: disable, 1: enable
                                       *only supported by H/W prior to Cx

       bit5            0       RW      on-pad fix horizontal scrolling enable
                                       0: disable, 1: enable
                                       *only supported by H/W prior to Cx

0x8290                         RW      software control register 1
       bit0            0       RW      absolute coordination mode
                                       0: disable, 1: enable
                                       *supported since Cx

       bit1            0       RW      gesture ID output
                                       0: disable, 1: enable
                                       *supported since Cx

       bit2            0       RW      two fingers' coordinates output
                                       0: disable, 1: enable
                                       *supported since Cx

       bit3            0       RW      finger up one packet output
                                       0: disable, 1: enable
                                       *supported since Cx

       bit4            0       RW      absolute coordination continuous mode
                                       0: disable, 1: enable
                                       *supported since Cx

       bit6~bit5       00      RW      gesture group selection
                                       00: basic
                                       01: suite
                                       10: suite pro
                                       11: advanced
                                       *supported since Cx

       bit7            0       RW      Bx packet output compatible mode
                                       0: disable, 1: enable
                                       *supported since Cx
                                       *supported since Cx


0x833d                         RW      on-pad command control 1
       bit7            1       RW      on-pad command detection enable
```

```
                                    0: disable, 1: enable
                                    *supported since Cx

0x833e                      RW      on-pad command detection
       bit7           0     RW      on-pad command left button down tag
                                    enable. Works only in H/W based PS/2
                                    data packet mode.
                                    0: disable, 1: enable
                                    *supported since Cx
```

Walkera WK-0701 transmitter

Walkera WK-0701 transmitter is supplied with a ready to fly Walkera helicopters such as HM36, HM37, HM60. The walkera0701 module enables to use this transmitter as joystick

Devel homepage and download: http://zub.fei.tuke.sk/walkera-wk0701/

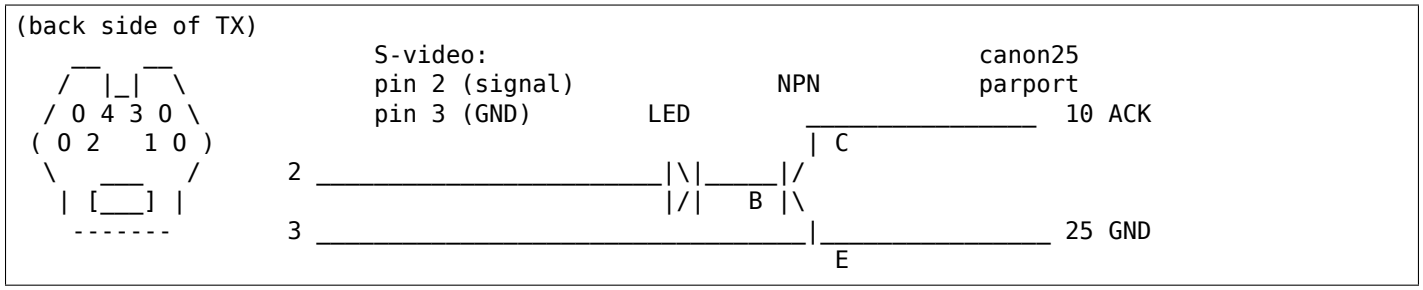
or use cogito: cg-clone http://zub.fei.tuke.sk/GIT/walkera0701-joystick

Connecting to PC

At back side of transmitter S-video connector can be found. Modulation pulses from processor to HF part can be found at pin 2 of this connector, pin 3 is GND. Between pin 3 and CPU 5k6 resistor can be found. To get modulation pulses to PC, signal pulses must be amplified.

Cable: (walkera TX to parport)

Walkera WK-0701 TX S-VIDEO connector:



I use green LED and BC109 NPN transistor.

Software

Build kernel with walkera0701 module. Module walkera0701 need exclusive access to parport, modules like lp must be unloaded before loading walkera0701 module, check dmesg for error messages. Connect TX to PC by cable and run jstest /dev/input/js0 to see values from TX. If no value can be changed by TX “joystick”, check output from /proc/interrupts. Value for (usually irq7) parport must increase if TX is on.

Technical details

Driver use interrupt from parport ACK input bit to measure pulse length using hrtimers.

Frame format: Based on walkera WK-0701 PCM Format description by Shaul Eizikovich. (downloaded from http://www.smartpropoplus.com/Docs/Walkera_Wk-0701_PCM.pdf)

Signal pulses

```
                  (ANALOG)
   SYNC      BIN   OCT
  +---------+      +------+
  |         |      |      |
--+         +------+      +---
```

Frame

```
SYNC , BIN1, OCT1, BIN2, OCT2 ... BIN24, OCT24, BIN25, next frame SYNC ..
```

pulse length

```
 Binary values:               Analog octal values:

 288 uS Binary 0              318 uS       000
 438 uS Binary 1              398 uS       001
                              478 uS       010
                              558 uS       011
                              638 uS       100
1306 uS SYNC                  718 uS       101
                              798 uS       110
                              878 uS       111
```

24 bin+oct values + 1 bin value = 24*4+1 bits = 97 bits

(Warning, pulses on ACK are inverted by transistor, irq is raised up on sync to bin change or octal value to bin change).

Binary data representations

One binary and octal value can be grouped to nibble. 24 nibbles + one binary values can be sampled between sync pulses.

Values for first four channels (analog joystick values) can be found in first 10 nibbles. Analog value is represented by one sign bit and 9 bit absolute binary value. (10 bits per channel). Next nibble is checksum for first ten nibbles.

Next nibbles 12 .. 21 represents four channels (not all channels can be directly controlled from TX). Binary representations are the same as in first four channels. In nibbles 22 and 23 is a special magic number. Nibble 24 is checksum for nibbles 12..23.

After last octal value for nibble 24 and next sync pulse one additional binary value can be sampled. This bit and magic number is not used in software driver. Some details about this magic numbers can be found in Walkera_Wk-0701_PCM.pdf.

Checksum calculation

Summary of octal values in nibbles must be same as octal value in checksum nibble (only first 3 bits are used). Binary value for checksum nibble is calculated by sum of binary values in checked nibbles + sum of octal values in checked nibbles divided by 8. Only bit 0 of this sum is used.

xpad - Linux USB driver for Xbox compatible controllers

This driver exposes all first-party and third-party Xbox compatible controllers. It has a long history and has enjoyed considerable usage as Window's xinput library caused most PC games to focus on Xbox controller compatibility.

Due to backwards compatibility all buttons are reported as digital. This only effects Original Xbox controllers. All later controller models have only digital face buttons.

Rumble is supported on some models of Xbox 360 controllers but not of Original Xbox controllers nor on Xbox One controllers. As of writing the Xbox One's rumble protocol has not been reverse engineered but in the future could be supported.

Notes

The number of buttons/axes reported varies based on 3 things:

- if you are using a known controller
- if you are using a known dance pad
- if using an unknown device (one not listed below), what you set in the module configuration for "Map D-PAD to buttons rather than axes for unknown pads" (module option dpad_to_buttons)

If you set dpad_to_buttons to N and you are using an unknown device the driver will map the directional pad to axes (X/Y). If you said Y it will map the d-pad to buttons, which is needed for dance style games to function correctly. The default is Y.

dpad_to_buttons has no effect for known pads. A erroneous commit message claimed dpad_to_buttons could be used to force behavior on known devices. This is not true. Both dpad_to_buttons and triggers_to_buttons only affect unknown controllers.

Normal Controllers

With a normal controller, the directional pad is mapped to its own X/Y axes. The jstest-program from joystick-1.2.15 (jstest-version 2.1.0) will report 8 axes and 10 buttons.

All 8 axes work, though they all have the same range (-32768..32767) and the zero-setting is not correct for the triggers (I don't know if that is some limitation of jstest, since the input device setup should be fine. I didn't have a look at jstest itself yet).

All of the 10 buttons work (in digital mode). The six buttons on the right side (A, B, X, Y, black, white) are said to be "analog" and report their values as 8 bit unsigned, not sure what this is good for.

I tested the controller with quake3, and configuration and in game functionality were OK. However, I find it rather difficult to play first person shooters with a pad. Your mileage may vary.

Xbox Dance Pads

When using a known dance pad, jstest will report 6 axes and 14 buttons.

For dance style pads (like the redoctane pad) several changes have been made. The old driver would map the d-pad to axes, resulting in the driver being unable to report when the user was pressing both left+right or up+down, making DDR style games unplayable.

Known dance pads automatically map the d-pad to buttons and will work correctly out of the box.

If your dance pad is recognized by the driver but is using axes instead of buttons, see section 0.3 - Unknown Controllers

I've tested this with Stepmania, and it works quite well.

Unknown Controllers

If you have an unknown xbox controller, it should work just fine with the default settings.

HOWEVER if you have an unknown dance pad not listed below, it will not work UNLESS you set "dpad_to_buttons" to 1 in the module configuration.

USB adapters

All generations of Xbox controllers speak USB over the wire.

- Original Xbox controllers use a proprietary connector and require adapters.
- Wireless Xbox 360 controllers require a 'Xbox 360 Wireless Gaming Receiver for Windows'
- Wired Xbox 360 controllers use standard USB connectors.
- Xbox One controllers can be wireless but speak Wi-Fi Direct and are not yet supported.
- Xbox One controllers can be wired and use standard Micro-USB connectors.

Original Xbox USB adapters

Using this driver with an Original Xbox controller requires an adapter cable to break out the proprietary connector's pins to USB. You can buy these online fairly cheap, or build your own.

Such a cable is pretty easy to build. The Controller itself is a USB compound device (a hub with three ports for two expansion slots and the controller device) with the only difference in a nonstandard connector (5 pins vs. 4 on standard USB 1.0 connectors).

You just need to solder a USB connector onto the cable and keep the yellow wire unconnected. The other pins have the same order on both connectors so there is no magic to it. Detailed info on these matters can be found on the net ([1], [2], [3]).

Thanks to the trip splitter found on the cable you don't even need to cut the original one. You can buy an extension cable and cut that instead. That way, you can still use the controller with your X-Box, if you have one ;)

Driver Installation

Once you have the adapter cable, if needed, and the controller connected the xpad module should be auto loaded. To confirm you can cat /sys/kernel/debug/usb/devices. There should be an entry like those:

Listing 3.1: dump from InterAct PowerPad Pro (Germany)

```
T:  Bus=01 Lev=03 Prnt=04 Port=00 Cnt=01 Dev#=  5 Spd=12  MxCh= 0
D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=32 #Cfgs=  1
P:  Vendor=05fd ProdID=107a Rev= 1.00
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=58(unk. ) Sub=42 Prot=00 Driver=(none)
E:  Ad=81(I) Atr=03(Int.) MxPS=  32 Ivl= 10ms
E:  Ad=02(O) Atr=03(Int.) MxPS=  32 Ivl= 10ms
```

[1] http://euc.jp/periphs/xbox-controller.ja.html (ITO Takayuki)
[2] http://xpad.xbox-scene.com/
[3] http://www.markosweb.com/www/xboxhackz.com/

Listing 3.2: dump from Redoctane Xbox Dance Pad (US)

```
 T:  Bus=01 Lev=02 Prnt=09 Port=00 Cnt=01 Dev#= 10 Spd=12  MxCh= 0
 D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
 P:  Vendor=0c12 ProdID=8809 Rev= 0.01
 S:  Product=XBOX DDR
 C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
 I:  If#= 0 Alt= 0 #EPs= 2 Cls=58(unk. ) Sub=42 Prot=00 Driver=xpad
 E:  Ad=82(I) Atr=03(Int.) MxPS=  32 Ivl=4ms
 E:  Ad=02(O) Atr=03(Int.) MxPS=  32 Ivl=4ms
```

Supported Controllers

For a full list of supported controllers and associated vendor and product IDs see the xpad_device[] array[4].

As of the historic version 0.0.6 (2006-10-10) the following devices were supported:

```
original Microsoft XBOX controller (US),    vendor=0x045e, product=0x0202
smaller  Microsoft XBOX controller (US),    vendor=0x045e, product=0x0289
original Microsoft XBOX controller (Japan), vendor=0x045e, product=0x0285
InterAct PowerPad Pro (Germany),            vendor=0x05fd, product=0x107a
RedOctane Xbox Dance Pad (US),              vendor=0x0c12, product=0x8809
```

Unrecognized models of Xbox controllers should function as Generic Xbox controllers. Unrecognized Dance Pad controllers require setting the module option 'dpad_to_buttons'.

If you have an unrecognized controller please see 0.3 - Unknown Controllers

Manual Testing

To test this driver's functionality you may use 'jstest'.

For example:

```
> modprobe xpad
> modprobe joydev
> jstest /dev/js0
```

If you're using a normal controller, there should be a single line showing 18 inputs (8 axes, 10 buttons), and its values should change if you move the sticks and push the buttons. If you're using a dance pad, it should show 20 inputs (6 axes, 14 buttons).

It works? Voila, you're done ;)

Thanks

I have to thank ITO Takayuki for the detailed info on his site http://euc.jp/periphs/xbox-controller.ja.html.

His useful info and both the usb-skeleton as well as the iforce input driver (Greg Kroah-Hartmann; Vojtech Pavlik) helped a lot in rapid prototyping the basic functionality.

[4] http://lxr.free-electrons.com/ident?i=xpad_device

References

Historic Edits

2002-07-16 - Marko Friedemann <mfr@bmx-chemnitz.de>

- original doc

2005-03-19 - Dominic Cerquetti <binary1230@yahoo.com>

- added stuff for dance pads, new d-pad->axes mappings

Later changes may be viewed with 'git log Documentation/input/xpad.txt'

Driver documentation for yealink usb-p1k phones

Status

The p1k is a relatively cheap usb 1.1 phone with:

- keyboard full support, yealink.ko / input event API
- LCD full support, yealink.ko / sysfs API
- LED full support, yealink.ko / sysfs API
- dialtone full support, yealink.ko / sysfs API
- ringtone full support, yealink.ko / sysfs API
- audio playback full support, snd_usb_audio.ko / alsa API
- audio record full support, snd_usb_audio.ko / alsa API

For vendor documentation see http://www.yealink.com

keyboard features

The current mapping in the kernel is provided by the map_p1k_to_key function:

```
Physical USB-P1K button layout       input events


          up                               up
    IN           OUT                 left,   right
         down                             down

  pickup   C    hangup               enter, backspace, escape
    1      2      3                   1, 2, 3
    4      5      6                   4, 5, 6,
    7      8      9                   7, 8, 9,
    *      0      #                   *, 0, #,
```

The "up" and "down" keys, are symbolised by arrows on the button. The "pickup" and "hangup" keys are symbolised by a green and red phone on the button.

LCD features

The LCD is divided and organised as a 3 line display:

```
  |[]   [][]   [][]   [][]   in   |[][]
  |[] M [][] D [][] : [][]   out  |[][]
                            store

  NEW REP         SU MO TU WE TH FR SA

  [] [] [] [] [] [] [] [] [] [] [] []
  [] [] [] [] [] [] [] [] [] [] [] []


Line 1  Format (see below)    : 18.e8.M8.88...188
        Icon names            :   M  D  :  IN OUT STORE
Line 2  Format                : .........
        Icon name             : NEW REP SU MO TU WE TH FR SA
Line 3  Format                : 888888888888
```

Format description: From a userspace perspective the world is separated into “digits” and “icons”. A digit can have a character set, an icon can only be ON or OFF.

Format specifier:

```
'8' :  Generic 7 segment digit with individual addressable segments

Reduced capability 7 segment digit, when segments are hard wired together.
'1' : 2 segments digit only able to produce a 1.
'e' : Most significant day of the month digit,
      able to produce at least 1 2 3.
'M' : Most significant minute digit,
      able to produce at least 0 1 2 3 4 5.

Icons or pictograms:
'.' : For example like AM, PM, SU, a 'dot' .. or other single segment
      elements.
```

Driver usage

For userland the following interfaces are available using the sysfs interface:

```
/sys/.../
         line1        Read/Write, lcd line1
         line2        Read/Write, lcd line2
         line3        Read/Write, lcd line3

         get_icons    Read, returns a set of available icons.
         hide_icon    Write, hide the element by writing the icon name.
         show_icon    Write, display the element by writing the icon name.

         map_seg7     Read/Write, the 7 segments char set, common for all
                      yealink phones. (see map_to_7segment.h)

         ringtone     Write, upload binary representation of a ringtone,
                      see yealink.c. status EXPERIMENTAL due to potential
                      races between async. and sync usb calls.
```

lineX

Reading /sys/../lineX will return the format string with its current value.

Example:

```
cat ./line3
888888888888
Linux Rocks!
```

Writing to /sys/../lineX will set the corresponding LCD line.

- Excess characters are ignored.
- If less characters are written than allowed, the remaining digits are unchanged.
- The tab ‘t’and ‘n’ char does not overwrite the original content.
- Writing a space to an icon will always hide its content.

  Example:

  ```
  date +"%m.%e.%k:%M"  | sed 's/^0/ /' > ./line1
  ```

  Will update the LCD with the current date & time.

get_icons

Reading will return all available icon names and its current settings:

```
cat ./get_icons
on M
on D
on :
   IN
   OUT
   STORE
   NEW
   REP
   SU
   MO
   TU
   WE
   TH
   FR
   SA
   LED
   DIALTONE
   RINGTONE
```

show/hide icons

Writing to these files will update the state of the icon. Only one icon at a time can be updated.

If an icon is also on a ./lineX the corresponding value is updated with the first letter of the icon.

Example - light up the store icon:

```
echo -n "STORE" > ./show_icon

cat ./line1
18.e8.M8.88...188
              S
```

Example - sound the ringtone for 10 seconds:

```
echo -n RINGTONE > /sys/..../show_icon
sleep 10
echo -n RINGTONE > /sys/..../hide_icon
```

Sound features

Sound is supported by the ALSA driver: snd_usb_audio

One 16-bit channel with sample and playback rates of 8000 Hz is the practical limit of the device.

Example - recording test:

```
arecord -v -d 10 -r 8000 -f S16_LE -t wav  foobar.wav
```

Example - playback test:

```
aplay foobar.wav
```

Troubleshooting

Q Module yealink compiled and installed without any problem but phone is not initialized and does not react to any actions.

A If you see something like: hiddev0: USB HID v1.00 Device [Yealink Network Technology Ltd. VOIP USB Phone in dmesg, it means that the hid driver has grabbed the device first. Try to load module yealink before any other usb hid driver. Please see the instructions provided by your distribution on module configuration.

Q Phone is working now (displays version and accepts keypad input) but I can't find the sysfs files.

A The sysfs files are located on the particular usb endpoint. On most distributions you can do: "find /sys/ -name get_icons" for a hint.

Credits & Acknowledgments

- Olivier Vandorpe, for starting the usbb2k-api project doing much of the reverse engineering.
- Martin Diehl, for pointing out how to handle USB memory allocation.
- Dmitry Torokhov, for the numerous code reviews and suggestions.