Diploma Thesis

University of Applied Sciences Augsburg
Department of Computer Science

Input Abstraction Layer

Design and Implementation of an Extended Input Interface

Submitted by Timo Hönig, Winter Semester 2004/2005

Examiner:	Prof. Dr. Hubert Högl
Examiner:	Prof. Burkhard Stork
Supervisor:	Dipl. Inf. (univ.) Stefan Behlert

The Diploma Thesis has been created in cooperation with SUSE Linux AG, Nuremberg.

Diploma Thesis

University of Applied Sciences Augsburg
Department of Computer Science

I affirm that the diploma thesis is my own work and has never been used before, for any auditing purposes. All used sources, additional used information and citations are quoted as such.

Timo Hönig

Input Abstraction Layer

Timo Hönig

Timo Hönig, Johannes-Haag-Straße 2, D-86153 Augsburg

08 07 06 05 04   6 5 4 3 2

First edition:	15. December 2004
Second impression, with corrections:	9. January 2005

Contents

Chapter 1

An Introduction to Linux Input

The Linux kernel offers drivers for a vast number of different input devices—drivers for keyboards, mice, joysticks, touch screens and others. Even though, many of these input devices can not be used with the GNU/Linux operating system. This severe problem is caused by input device drivers' varying implementations: they are not following a common standard. It is in the nature of the process how free and open source development evolves by the course of time. Most developers implement device drivers in their sole discretion, only a few are following common approaches. The increasing number of input devices are connected in many different ways which results in a numerous amount of device drivers. For example, most function keys found on mobile computers follow an entirely different approach for processing input events than regular keyboards. All keys—no matter how they are connected to the system—are intended to trigger input events which can be utilized. However, applications are not prepared to receive input events from all the interfaces offered by the input device drivers.

The project outcome resulting from this diploma thesis solves the problem by creating an abstraction layer—the Input Abstraction Layer. The Input Abstraction Layer represents a framework unifying input events of the different input device drivers. All input events are abstracted and get delivered to user space applications. Applications using the framework receive all input events by accessing a single interface provided by the Input Abstraction Layer.

To get an overview of the current situation regarding input device support under Linux a survey was carried out (Appendix B). This survey was aimed at all Linux users: beginners, experts and professionals. The questions of the survey targeted function keys of desktop keyboards and special keys of built-in laptop keyboards. The target of the survey was to investigate the current state of the support of input devices from the user's point of view. It lasted for five days starting on the 13th of October 2004. During this time, 305 participants contributed to the survey by answering the questions posted. Since it was an online survey an e-mail was posted to several mailing lists. Both the e-mail and the addressed mailing lists can be found in Appendix B, §B.1.

The used set of mailing lists ensured that all target groups—Linux beginners, experienced and professionals—were reached. The Linux distributions' mailing lists are mainly subscribed to by end users, the Gnome and KDE lists are subscribed by both Linux developers and users. The laptop mailing lists are subscribed to by people who run Linux on laptops, and therefore, have to face extraordinary problems caused by the specific hardware. The large amount of participants who contributed to the survey together with the used set of mailing lists, resulted in a high expressiveness of the survey's evaluation.

1.1 Current State of Input Devices

The answers to the questions regarding desktop keyboards (Appendix B, §B.3) revealed that Linux beginners are not the only one unsatisfied with the current support for desktop keyboards. While 70% of the Linux beginners working on desktop systems are unhappy with the current situation, almost 50% of the experienced users demand better support for desktop keyboards, too. Professionals are more contented: about 42% are satisfied with the support for desktop keyboards. Looking at these results, it is apparent that even support for conventional desktop keyboards needs to be improved. The results regarding Linux support for desktop keyboards are shown in Figure 1.1.

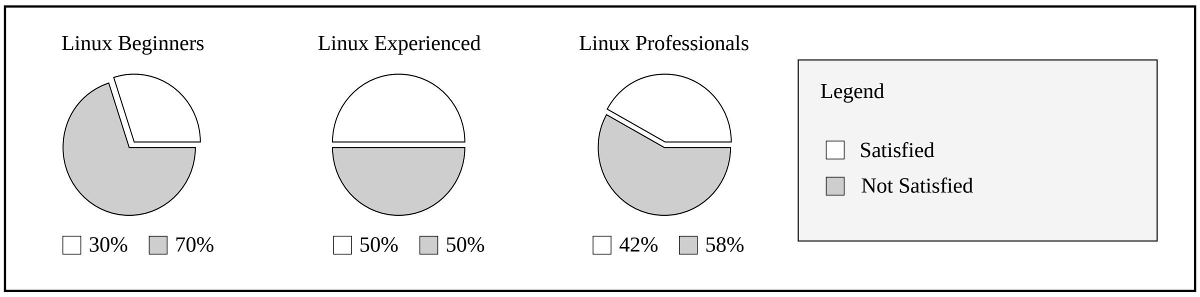


Figure 1.1: Linux Support for Desktop Keyboards

The results for built-in laptop keyboards (Appendix B, §B.4) are disappointing. All participants who classified themselves as beginners report that they face problems with the function keys found on their laptops. 33% of the Linux beginners can not use their function keys at all. The rest (67%) of the beginners stated that they can use at least some of them—but nobody reported that all function keys are working. 35% of the experienced Linux users working with laptops report that all function keys are working as supposed. More than half (52%) of the experienced users state that at least some of the function keys are working. For about 13%, none of the function keys are working. Only 7% of the professionals state that they can not use function keys of their laptops at all. 49% can use all of them and 43% are able to use some of the function keys.

It is apparent that one needs expertise to get function keys working on laptops. In most cases it is actually possible to use at least some of them. But due to missing interfaces users without profound knowledge about Linux are ignored. The results regarding Linux support for function keys of laptop keyboards are shown in Figure 1.2.

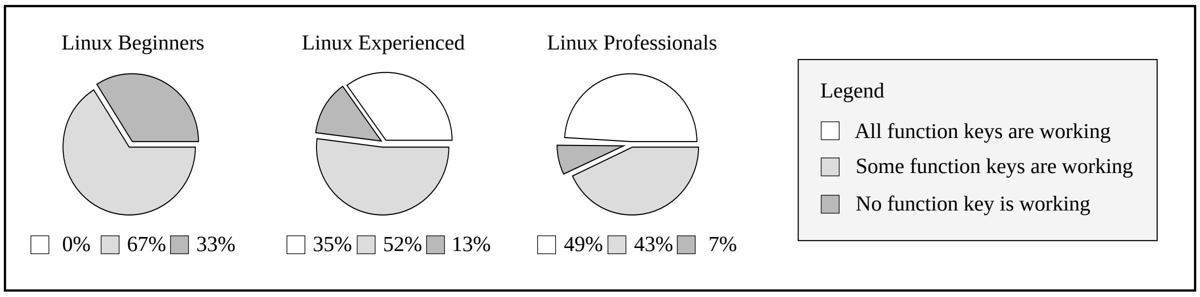


Figure 1.2: Linux Support for Laptop Keyboards

Recent laptops offer various features like dynamic CPU frequency and adjustment of screen brightness to save power consumption. Such features are commonly controlled by function keys which are—as shown above—in many instances not working on Linux systems. It is not unusual that functions like switching between LCD and CRT are not working, too. About half (48,6%) of all participants stated that only some, or even none, of these essential function keys are working on their system.

The questions regarding the demand of features (Appendix B, §B.5) evinced that many users (58%) would like to have on-screen display notifications (e.g. screen brightness, change of volume). The majority of the participants classify the possibility to switch between LCD and CRT screen either as "would like to have" or "must have". Other requested features include the ability to launch arbitrary applications (43%), the chance to change screen brightness (62%) and the possibility to change the system's volume with the supposed function keys (72%).

1.2 Objective

The objective of this diploma thesis is to close the gap between the existing Linux input device drivers and the user interface. As shown by the survey, experienced users demand better support for input devices just as beginners do. This implies that whether input devices can be utilized or not is not only a question of knowledge.

Since the different input device drivers are accessed in different ways, and since their data structures for input events differ from each other, there is an urgent need for the Input Abstraction Layer. The Input Abstraction Layer needs to gather as many events as possible and deliver them—no matter which interface the event originally came from—in a unified data representation to user space. Assuming that, applications get notice of user input they were not able to receive before.

1.3 Overview of the Diploma Thesis

The chapters are build on top of each other. Chapter 2 discusses Linux input device drivers and their interfaces. It is followed by three chapters discussing the requirements specification, architecture, design and the actual implementation of the Input Abstraction Layer. The achievements are discussed in Chapter 6 by chiefly analyzing whether the specified requirements are met by the implementation. Chapter 7 explains the development methodology of Free and Open Source Software on the basis of the Input Abstraction Layer. The last chapter retrospects the diploma thesis as a whole and shows directions for the future of the Input Abstraction Layer.

Many passages of Chapter 2 and 5 discuss source code written in C. The reader is required to be acquainted with the C programming language to comprehend these passages. Additionally, expertise with common Unix tools and libraries is helpful. All chapters reference further reading, which are summarized in the bibliography. The following typographic conventions are used:

italic	Used for numeric variables, values of variables and to emphasize terms.
`constant width`	Used for commands, files, directories, options, programs, source code and variable names.

Throughout the text the notation `term(n)` indicates that a reference manual is available for `term`. The value n determines the type of `term`:

– `1`: Executable programs or shell commands
– `2`: System calls (functions provided by the kernel)
– `3`: Library calls (functions within program libraries)

E.g. `open(2)` indicates that `open` is a system call. The referenced manual can be read using the command `man 2 open` in a regular Linux console.

This diploma thesis has been realized by using free and open source software only. A list of the applications can be found in Appendix A. Debian GNU/Linux and SUSE Linux 9.2 running Linux kernel version 2.6.9 have been used for both the development of the Input Abstraction Layer and the typesetting of this text. The Input Abstraction Layer project, as a result of this diploma thesis is released under open source licenses (Chapter 7, §7.1). The diploma thesis itself is released under the terms of the Creative Commons Attribution License. Everyone is free to copy, distribute, display, and perform the work, to make derivative works and to make commercial use of the work. The only condition to be fulfilled is that the original author must be credited. A summary of the Creative Commons Attribution License and the licenses' legal code are found in Appendix E.

Chapter 2

Drivers and Interfaces

This chapter discusses the available input device drivers and their user space interfaces. Several parts of the Linux kernel source code are examined to give both a decent understanding and an overview. Current issues regarding user input are addressed where applicable. All relevant kernel configuration options are described and mentioned at appropriate passages. As further reading, [Low04] deepens the configuration and compilation of the Linux kernel.

Several sections of this chapter are discussing source code of the Linux kernel. These excerpts are based on version 2.6.9 of the Linux kernel. The mentioned path names are relative to the root of the source tree of the Linux kernel (Appendix D, §D.1). A practical look at the design and implementation of the Linux version 2.6 is given in [Lov03]. The architecture of the Linux kernel version 2.6 is comprehensively described in [Mau03].

The first section of this chapter discusses the Linux input core which is the most important driver for input devices: it connects input device drivers (e.g. keyboard drivers, §2.2) with input event handlers (e.g. event interface, §2.5). The event handlers are responsible to deliver occurring input events to both user space and kernel space. Section §2.3 and Section §2.4 describe Linux support for input devices which are connected using either USB or Bluetooth. In general, USB and Bluetooth input devices depend on the Linux input core, too. The Advanced Configuration and Power Interface (ACPI) defines several buttons which trigger input events. Section §2.6 discusses the Linux ACPI implementation concerning ACPI events. Input interfaces offered by specific input device drivers are discussed in Section §2.7 at the end of the chapter.

2.1 Input Core

The most significant driver regarding input devices is the input core driver `input.c`, which is found in the directory `drivers/input`. This driver chains input device drivers with input event handlers and implements the generic input interface for Linux input devices. The input device drivers communicate with the hardware and report occurring events to the input core. Consecutively, the events are passed on to the event handlers by the input core driver. The event handlers subsequently distribute the input events to different interfaces—user space interfaces as well as kernel space interfaces. The cooperation of input devices, their drivers, the input core and the event handlers is shown in Figure 2.1.

Beside others, the input core implements the functions `input_register_device()` and `input_unregister_device()`. An input device driver needs to register itself by calling the

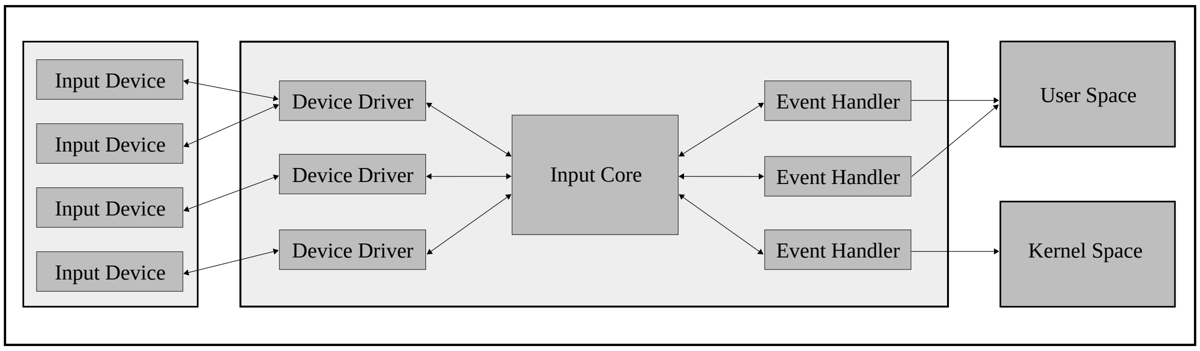


Figure 2.1: Linux Input System

function `input_register_device()` first. The driver calling `input_register_device()` must supply a pointer to a structure `input_dev` as parameter.

```
void input_register_device(struct input_dev *);
```

The structure `input_dev` is defined in `<linux/input.h>` and needs to be setup by the input driver itself. It contains information about the input device's capabilities and function pointers required by the Linux kernel in order to access the device. Once a driver is loaded, the input core initializes the timer of `input_dev` and adds the new input device to a list. If the kernel has enabled the hotplug subsystem (`CONFIG_HOTPLUG`), the input core also notifies the hotplug subsystem about the newly added device.

The function `input_unregister_device()` is called if an input device driver is removed. The input core removes the device drivers' data structures. Again, if hotplug is enabled, the hotplug subsystem will be notified about the removal of the device. Event handlers register themselves by calling the input core's function `input_register_handler()`. Upon registration, an event handler receives input events as a reference to a structure `input_event` from the input core. The structure `input_event` represents an input event and is defined in `<linux/input.h>`:

```
struct input_event {
     struct timeval time;
     __u16 type;
     __u16 code;
     __s32 value;
};
```

The member `time` contains a timestamp when the event happened. All valid event types, codes and values are defined in `<linux/input.h>`. For example, a keypress event and a key release event have the same `type` and `code` but a different `value`. For both events, `type` is set to `EV_KEY`. The value of `code` is unique for each key and contains the scancode. If the event was a keypress `value` equals *1*, if the event was a key release `value` equals *0*.

The timestamp `time` is generated by the input core. The other members of the struct—`type`, `code` and `value`—are supplied by the calling input device driver. Both the data structure of an input event and the function, which is invoked by an input device driver to report an event to the input core, are named `input_event`. Once an input device driver has reported an event by calling `input_event()`, the input core generates a structure `input_event` which is passed to the registered input event handlers. The function `input_event()` is defined in `<linux/input.h>`.

```
void input_event(struct input_dev *dev, unsigned int type,
                 unsigned int code, int value);
```

Instead of calling `input_event()`, device drivers can use one of several wrapper functions; for the most commonly used event types (e.g. `EV_KEY`) the input core provides wrapper functions which call `input_event()` with a hard-coded value for `type`. For example:

```
static inline void input_report_key(struct input_dev *dev, unsigned int code,
                                     int value)
{
        input_event(dev, EV_KEY, code, !!value);
}
```

In this case, the wrapper function `input_report_key()` not only calls `input_event()` but also normalizes `value` to either *0* or *1*.

The input device support for the Linux kernel is enabled by setting the configuration option `CONFIG_INPUT` to either *y* (compile static into kernel) or *m* (compile as module). Setting `CONFIG_INPUT=`*n* would result in a system which can not be controlled by a keyboard or another input device. Thus, the default is `CONFIG_INPUT=`*y*.

2.2 Keyboard Device Drivers

Drivers for desktop keyboards which are not connected using either USB or Bluetooth are located in `drivers/input/keyboard`. All keyboard drivers register themselves with the function `input_register_device()` as described above and report occurring events by calling the input core's function `input_event()` or respectively one of the wrapper functions.

Beside the driver for AT and PS/2 keyboards, Linux offers drivers for the following keyboards: Sun Type 4 and Type 5, DECstation/VAXstation LK201/LK401, IBM PC/XT connected by a parallel port keyboard adapter, the Apple Newton keyboard, keyboards connected via the Maple bus found on the game console Dreamcast and Amiga keyboards. These drivers are indicated as "Device Drivers" in Figure 2.1.

The corresponding event handler for input events generated by the various keyboard drivers is `keyboard.c`, which is found in the directory `drivers/char`. All keyboard drivers depend on this event handler. Upon connecting a new input device to the system the function `kbd_connect()` is called. This function is implemented by `keyboard.c` and validates the new device for its capabilities. If the device has the ability to generate keyboard input events, `keyboard.c` accepts this device by adding itself as an event handler for the input device and therefore handles its future input events.

Since the keyboard device drivers are reporting scancodes to the input core, the event handler `keyboard.c` is responsible to convert the scancodes to corresponding keycodes. For each keypress a keyboard generates a sequence of scancodes which result in a series of input events. The input events carry the scancodes, the event handler `keyboard.c` translates the scancodes to their corresponding keycodes. This mapping is necessary to operate one kind of keyboard with multiple languages. Only the keys' imprint differs—the scancode of each key remains the same.

Useful tools to debug scan- and keycodes are `showkey(1)`, `loadkeys(1)`, `dumpkeys(1)` and `setkeycodes(8)`. The source package of these utilities is available from `http://lct.sourceforge.net/`. The keymaps found in `/usr/share/keymaps` are translation tables—from scancode to keycode—and can be loaded by `loadkeys(1)` in user space.

Linux support for various keyboards is enabled by setting their configuration option to either *y* (compile static into kernel) or *m* (compile as module):

- `KEYBOARD_ATKBD` (AT and PS/2 keyboards)
- `KEYBOARD_SUNKBD` (Sun Type 4 and Type 5 keyboards)
- `KEYBOARD_LKKBD` (DECstation/VAXstation LK201/LK401 keyboards)
- `KEYBOARD_XTKBD` (IBM PC/XT keyboard)
- `KEYBOARD_NEWTON` (Apple Newton keyboard)
- `KEYBOARD_MAPLE` (Dreamcast keyboard)
- `KEYBOARD_AMIGA` (Amiga keyboard)

2.3 USB Input Device Drivers

The USB specification for Human Interface Devices (HID) describes which USB devices should be considered as HID [Usb01]:

- Keyboards and pointing devices, e.g. standard mouse devices, trackballs and joysticks
- Front-panel controls, e.g. knobs, switches, buttons and sliders
- Controls that might be found on devices such as telephones, remote controls, games or simulation devices, e.g. data gloves, throttles, steering wheels and rudder pedals
- Devices that may not require human interaction but provide data in a similar format to HID class devices, e.g. bar-code readers, thermometers and voltmeters

Due to the USB HID specification, there is only a need for a single device driver for all human interface devices instead of one device driver for each human interface device. This is one of the facts that led to the success of USB—especially regarding Linux since manpower for driver development is often missing.

Figure 2.2 shows how USB input devices and the Linux input core cooperate. USB devices are connected to the USB host controller (shown as "USB HC") which is driven by the corresponding host controller driver ("USB HCI"). Support for USB keyboards, mice and joysticks is implemented by the drivers HID core (`hid-core.c`) and HID input (`hid-input.c`) which are both located in the directory `drivers/usb/input`. The HID input driver uses the Linux input core and therefore depends on it. As before, new input devices are registered by calling `input_register_device()` and input events are reported by calling the input core's function `input_event()` or one of its wrapper functions. Some USB device drivers are not covered by

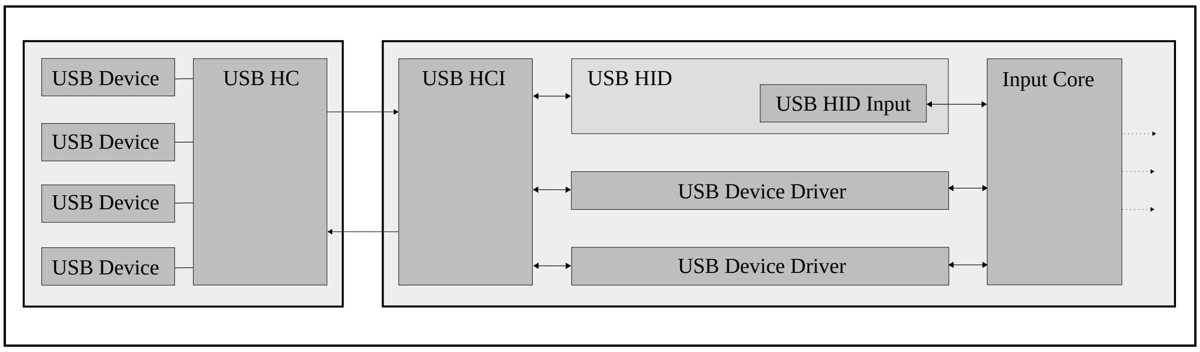


Figure 2.2: Linux USB Input

USB HID and USB HID input but use the input core driver, too. They are indicated as "USB Device Driver" in Figure 2.2.

USB support for the Linux kernel is enabled by setting the configuration option `CONFIG_USB` to either *y* (compile static into kernel) or *m* (compile as module). In addition, it is mandatory to choose a driver for the USB host controller. Linux supports the host controllers EHCI (USB 2.0), OHCI and UHCI. Their corresponding configuration options are `CONFIG_EHCI`, `CONFIG_OHCI` and `CONFIG_UHCI`. To enable USB HID support the configuration options `USB_HID` and `USB_HIDINPUT` have either to be set to *y* or *n*.

2.4 Bluetooth Input Device Drivers

The Bluetooth specification [Sig04] defines several *profiles*. Each profile represents a class of similar Bluetooth devices. Bluetooth input devices are referred to the HID profile, which is specified in [RG03]. Both the USB HID specification and the Bluetooth HID specification are very similar and list the same type of devices which are considered as HID. Again, this fact is important to support future devices without developing a new driver for every new device.

Due to the fact that Bluetooth host controllers can be attached in several ways, Figure 2.3 shows the different connection methods. It is possible to use USB, PCMCIA or UART (serial connection) to connect a Bluetooth host controller. There are also Bluetooth host controllers which use a serial connection via PCMCIA. This method is not shown in Figure 2.3.

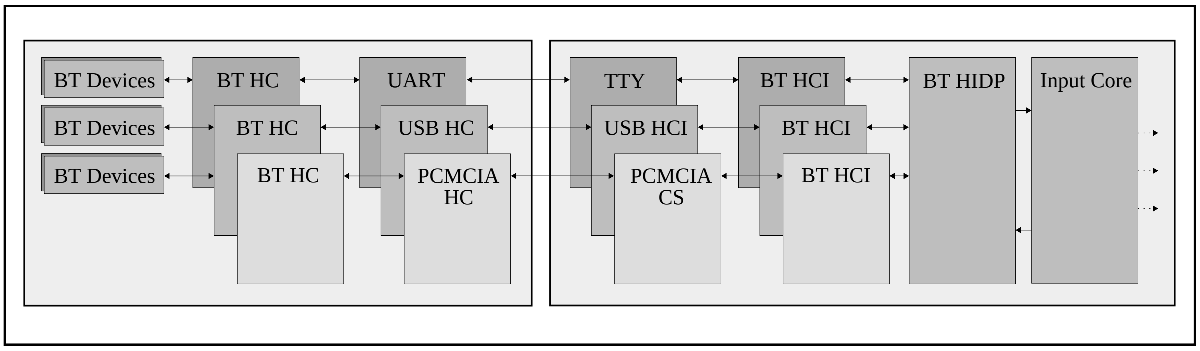


Figure 2.3: Linux Bluetooth Input

Bluetooth devices ("BT Devices") are connected to a Bluetooth host controller ("BT HC") which is attached to the system as described above. The host controllers are driven by the corresponding Bluetooth host controller drivers. The Linux Bluetooth protocol stack BlueZ (`http://www.bluez.org`) implements the Bluetooth subsystem consisting of the Bluetooth Core, the Bluetooth host controller interface ("BT HCI") and several modules which are used to implement the various Bluetooth profiles. The Bluetooth HID profile is implemented by the Human Interface Device Protocol ("HIDP"). Bluetooth HID are registered by the HIDP driver at the Linux input core. Once again, this is done by calling input core's function `input_register_device()`. HIDP depends on the Logical Link Control and Adaptation Protocol (L2CAP). This protocol provides a layer for connection-oriented and connectionless data transport to Bluetooth devices.

In contrast to USB, connections between Bluetooth devices and the host controller are initiated from user space. Utilities are available from `http://www.bluez.org/download.html`.

To enable Bluetooth support for the Linux kernel it is necessary to set the configuration option `CONFIG_BT` to either *y* (compile static into kernel) or *m* (compile as module). Bluetooth HID devices require HIDP (`CONFIG_BT_HIDP`) which depends on L2CAP support (`CONFIG_BT_L2CAP`). At the time of writing the following Bluetooth host controllers are supported by Linux:

- USB: Bluetooth host controllers with USB interface (`CONFIG_BT_HCIUSB`), AVM BlueFRITZ! USB (`CONFIG_BT_HCIBFUSB`)
- PCMCIA or Compact Flash: Nokia DTL1 (`CONFIG_BT_HCIDTL1`), 3Com BT3C used on 3Com Bluetooth Card and HP Bluetooth Card (`CONFIG_BT_HCIBT3C`), Anycom BlueCard (`CONFIG_BT_HCIBLUECARD`) and Xircom CreditCard Bluetooth Adapter, Xircom RealPort2 Bluetooth Adapter, Sphinx PICO Card, H-Soft blue+Card, Cyber-blue Compact Flash Card (`CONFIG_BT_HCIBTUART`)
- UART: Bluetooth host controllers with a serial port interface (`CONFIG_BT_HCIUART`)

2.5 Event Interface

Linux offers a generic event interface which gives user space access to input devices. This event interface is implemented by `evdev.c` which is found in the directory `drivers/input`. For each input device which uses the input core, the event interface registers at least one device node in `/dev/input/`. The devices allocated by the event interface are named `event`*n* with *n* running from *0* to `EVDEV_MINORS`−`1`. By default, `EVDEV_MINORS` equals *32* which means that up to 32 input devices can be handled by the event interface, namely `event0`... `event31`.

The event interface driver registers itself as an event handler using the input core's function `input_register_handler()`. Input events from the input device drivers are passed to the corresponding `event`*n* device. User space applications can read input events by opening a file descriptor for `/dev/input/event`*n* (e.g. using `open(2)`) and afterwards performing `read(2)` on the file descriptor. In case of an input event, applications reading from `/dev/input/event`*n* receive structures of the type `input_event`. It is possible to use both blocking and non-blocking reads on the `/dev/input/event`*n* device nodes. Furthermore, it is possible to perform operations on input devices using `ioctl(2)`. The valid `ioctl(2)` requests for the event interface are defined in `<linux/input.h>`.

To enable the event interface the Linux kernel has to be compiled with `INPUT_EVDEV=`*y* or `INPUT_EVDEV=`*m*. The driver `evbug.c` (`INPUT_EVBUG`) can be used to debug the event interface. If event debugging is enabled, all input events are printed to the system log via `printk()`.

2.6 Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) specification [Acp04] defines different buttons as user input: a power button, a sleep button and a lid switch. While the power button is mandatory according to the ACPI specification, the sleep button and the lid switch are both optional. The lid switch only applies to mobile computers. The operating system is responsible for the action which is taken upon receiving an event from one of the buttons: it may invoke suggestive functions of the operating system's ACPI implementation, such as switching the system to a sleep or power off state, but it also may ignore the button events at all.

The Linux ACPI implementation exports several interfaces to user space using the virtual file system `proc`, usually mounted at `/proc`. The ACPI bus driver (`bus.c`, located in `drivers/acpi`) creates the ACPI root entry `/proc/acpi`. ACPI drivers which want to offer an interface in user space create `proc` entries underneath this root entry. Most of these entries are read-only. User space applications can use those interfaces to gather information. Some ACPI drivers export interfaces which can be used for both gathering information and invoking functions of the ACPI subsystem.

Most ACPI drivers are using the function `acpi_bus_generate_event()` to report events. This function is provided by the ACPI bus driver. Besides button events (`button.c`), ACPI events include notifications about the thermal state of devices (`thermal.c`), the charging state of batteries (`battery.c`), the state of the AC adapter (`ac.c`) and events issued by CPUs (`processor.c`). However, the button events are the only events which can directly be invoked by the user.

The interface for ACPI events (read-only) is realized by the driver `event.c`. User space applications can access this interface by reading `/proc/acpi/event`. If no user space process is actually reading from the event interface, the ACPI events are not written to `/proc/acpi/event` by the event driver. Whether a process is reading from the interface or not is determined by checking the value of the variable `event_is_open`. As soon as a user space process invokes the system call `open(2)` in order to access `/proc/acpi/event`, the event driver's function `acpi_system_open_event()` is invoked. This function sets `event_is_open` to *1*. Correspondingly, `event_is_open` is set to *0* by `acpi_system_close_event()` once the process stops reading from the event interface.

ACPI events are volatile as long as no user space process is accessing the event interface. But once a user space process reads from `/proc/acpi/event`, it receives the ACPI events that occur, since the file descriptor was opened by the process. As read access to `/proc/acpi/event` can not be concurrent, a user space application is responsible for the distribution of the ACPI events. Drivers generating ACPI events call the bus driver's function `acpi_bus_generate_event()`:

```
int acpi_bus_generate_event(struct acpi_device *device, u8 type, int data);
```

The calling function has to pass a pointer `device` to the ACPI device which has triggered the event, an event type (`type`) and event data (`data`) as parameters. Only in case of a process reading from the ACPI event interface, the bus driver generates the actual ACPI event. An ACPI event has the following data structure (defined in `<acpi/acpi_bus.h>`):

```
struct acpi_bus_event {
     struct list_head node;
     acpi_device_class device_class;
     acpi_bus_id bus_id;
     u32 type;
     u32 data;
};
```

To avoid race conditions, ACPI events are managed using the double linked list: if an ACPI event occurs while another ACPI event is still being processed by the ACPI event driver, the ACPI bus driver appends the new event to the tail of the list `acpi_bus_event_list`. The member `device_class` is a char array which gives information about the device class. While the ACPI device drivers are responsible to determine the device class for each ACPI device, the ACPI bus driver copies this information to the ACPI event structure. The same procedure applies to `bus_id`. The members `type` and `data` contain additional information about the actual event.

Button Driver

If the user presses one of the ACPI buttons and if the lid gets opened or closed a corresponding ACPI event is generated by the ACPI button driver. The device class `device_class` is set to *button/power* for the power button, *button/sleep* for the sleep button and *button/lid* for the lid switch. Correspondingly, the value of `bus_id` is set to *PWRF* for the power button, *SLPF* for the sleep button and *LID* for the lid switch. For all button events, `type` is set to *0x80*. Chapter 5.6.3 (Device Object Notifications) of the ACPI specification [Acp04] defines the valid values for notifications. The member `data` is a counter and gets incremented each time the corresponding button is pressed.

For example, a process reading from the ACPI event interface receives the following output if first the power button is pressed, followed by pressing the sleep button and finally the lid is closed and opened again.

```
button/power PWRF 00000080 00000001
button/sleep SLPF 00000080 00000001
button/lid LID 00000080 00000001
button/lid LID 00000080 00000002
```

While the events for the power button, sleep button and lid switch are common for all ACPI compliant systems, there are several drivers for specific mobile computers which report input events of special or function keys using ACPI, too. These drivers either use the ACPI bus driver's function `acpi_bus_generate_event()`—and therefore generate events which can be received using the ACPI event interface—or by implementing their own interface. Input event drivers using ACPI which are part of the Linux kernel are discussed next, other ACPI drivers which are available separately are discussed in Section §2.7.

Input Event Drivers

Drivers which are using the ACPI bus driver to report input events are the Asus/Medion ACPI driver (`asus_acpi.c`) and the IBM ThinkPad ACPI driver (`thinkpad_acpi.c`). On supported systems the Asus/Medion driver generates ACPI events with `device_class` set to *hotkey* and `bus_id` set to *HOTK*. The value for `type` depends on the key pressed. As previously mentioned `data` represents how often the corresponding key is pressed. The IBM ThinkPad ACPI driver generates events with `device_class` set to *ibm/hotkey* and `bus_id` set to *HKEY*. While `type` is always set to *0x80*, the value of *data* depends on which key is pressed. A list of the systems supported by the Asus/Medion and IBM ThinkPad ACPI drivers is found in Appendix C, §C.2 and §C.5.

The Toshiba ACPI driver `toshiba_acpi.c` does not use `acpi_bus_generate_event()` to report events. Toshiba is using proprietary technology to implement the function keys. This proprietary hardware control interface (HCI) is used on recent Toshiba laptops. As soon as a function key is pressed a unique value for the pressed key is stored in a specific register. This register is called *system event register*. Since no interrupt occurs whenever a function key is pressed, the system event register needs to be polled. This polling is not done in kernel space. The Toshiba ACPI driver exports a read-write interface to user space. The location of the interface is `/proc/acpi/toshiba/keys`. The read and write operations of user space process on this interface invoke corresponding functions of the Toshiba ACPI driver, which is then able to access the HCI. A list of the systems supported by the Toshiba ACPI driver is found in Appendix C, §C.8.

The system event register is a FIFO register and can store values for up to 16 function key events. A user space process reading from `/proc/acpi/toshiba/keys` receives two values: `hotkey_ready` and `hotkey`. If `hotkey_ready` equals *0* it means that only one—the last—function key event is stored in the system event register. If there is more than one function key event stored in the system event register, `hotkey_ready` equals *1*. The value of `hotkey` is set to the value of the first element—the head—of the system event register. A user space process has to write *hotkey:1* to `/proc/acpi/toshiba/keys` in order to flush the first element of the FIFO register. Polling of the system event register can thereby be implemented in user space using the system calls `read(2)` and `write(2)`.

Subsystem Overview

Figure 2.4 shows the parts of the ACPI subsystem which are related to user input. The addressed drivers for mobile computers are pooled as "Other ACPI Drivers". The virtual file system `proc` is used to implement user space interfaces. The interface `/proc/acpi/`*other* is a wildcard for all ACPI input interfaces but the event interface (`/proc/acpi/event`).

The Linux kernel has to be compiled with the configuration option `CONFIG_ACPI` set to *y* to enable the ACPI core subsystem. It is not possible to compile ACPI support as module. However, all other discussed ACPI drivers can be either compiled statically or as module by setting the corresponding configuration option to either *y* or *m*:

- `CONFIG_ACPI_BUTTON` (ACPI Button Driver)
- `CONFIG_ACPI_ASUS` (Asus/Medion ACPI Driver)
- `CONFIG_ACPI_THINKPAD` (IBM ThinkPad ACPI Driver)
- `CONFIG_ACPI_TOSHIBA` (Toshiba ACPI Driver)

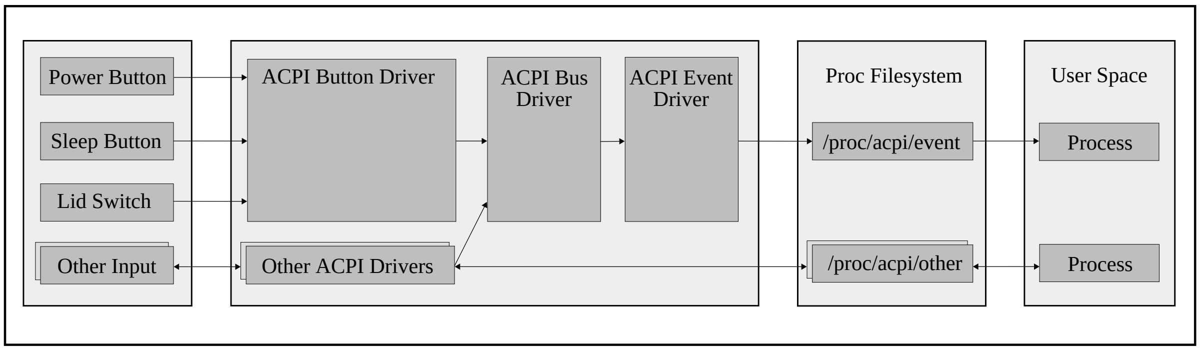


Figure 2.4: Linux ACPI Input

The event driver is part of the ACPI subsystem and does not need to be enabled explicitly in the kernel configuration. As mentioned above, user space is responsible for the distribution of the ACPI events. For this purpose, the ACPI daemon `acpid` (`http://acpid.sourceforge.net/`) is used by most Linux distributions. The ACPI daemon permanently reads `/proc/acpi/event` and distributes the events using Unix domain sockets to other user space applications. Besides distributing the events the ACPI daemon can be used to react upon occurring events.

2.7 Other Interfaces

Several other drivers for function keys of mobile computers are also available, and have not yet been discussed. Since many laptop vendors recently changed the hardware interface for function keys using ACPI, the following drivers mostly apply to laptops which are not recent. While some of these drivers are directly included in the mainline version of the Linux kernel, others are only available as kernel patch. The Linux kernel includes drivers for the following mobile computers: Dell (`CONFIG_DELL`), IBM ThinkPad (`CONFIG_NVRAM`), Sony (`CONFIG_SONYPI`) and Toshiba (`CONFIG_TOSHIBA`). Mobile computers from Acer, Hewlett Packard and Panasonic are supported by external drivers (not included in the Linux mainline kernel).

Many Dell laptops (Appendix C, §C.3) are supported by the Dell system management mode driver `i8k.c`, which provides the user space interface `/proc/i8k`. The `i8tools` package (available from `http://people.debian.org/~dz/i8k/`) requires this interface and contains a tool called `i8kbuttons`. This tool enables the use of the function keys found on the supported Dell laptops.

Function keys of IBM ThinkPad laptops can be used with the help of a user space application called `tpb` which stands for "ThinkPad Button". The NVRAM interface `/dev/nvram`—which is enabled by compiling the Linux kernel with the configuration option `CONFIG_NVRAM`—is required in order to use `tpb`. A list of the laptops which are supported by the driver is found in Appendix C, §C.6. The source package of `tpb` is available from `http://www.nongnu.org/tpb/`.

The "Panasonic Hotkey Driver" is available from `http://www.da-cha.org/letsnote/` and supports function keys found on several recent Panasonic laptops. On supported systems (Appendix C, §C.7), the driver generates events using the ACPI event interface. The driver generates events with `device_class` set to *pcc* and `bus_id` set to *HKEY*. While `type` is always set to *0x80*, the value of *data* depends on which hotkey is pressed. E.g. the hotkey combination Fn-F4 creates the following ACPI event:

```
pcc HKEY 00000080 00000084
```

Most Sony laptops have a proprietary programmable controller called “Sony Programmable I/O Control Device”. This device can be accessed using the Sony PI driver (`CONFIG_SONYPI`) which creates the user space interface `/dev/sonypi`. Additional user space programs are required to utilize this interface. Regarding user input, the interface provides access to information. Many Sony laptops have a so called “jogdial” which can be compared to the scroll wheel of a mouse. The jogdial can be moved in two directions (up and down) and it can be pressed. The driver also provides access to function keys which trigger events. The driver’s author offers user space tools which use the driver’s interface at `http://popies.net/sonypi/`.

In order to access the system management mode (SMM) on Toshiba laptops, the driver `toshiba.c` found in `drivers/char` is required. The driver creates the user space interface `/proc/toshiba` using the `proc` file system. Upon reading `/proc/toshiba`, the driver retrieves information and returns them to the reading process—for example, which function key was pressed at last. In addition to this interface, the Toshiba driver creates the device `toshiba` in `/dev` which can be used by applications to access the two proprietary interfaces SCI and HCI. The tool `toshset` (available from `http://www.schwieters.org/toshset/` is utilizing this interface. A list of the laptops supported by the Toshiba SMM driver is found in Appendix C, §C.9.

Function keys found on Acer laptops can be enabled using the “Acer Hotkey driver for Linux” (`http://www.informatik.hu-berlin.de/~tauber/acerhk/`). This driver directly accesses the system’s BIOS and generates input event using the Linux input core. Hence, no additional user space program is needed to use the function keys. A list of the laptops supported by the Acer driver is found in Appendix C, §C.1.

Most function keys found on Hewlett Packard laptops already trigger normal keyboard scancodes, thus they are covered by the Linux input core and do not need an additional driver or software. Some function keys are not working but can be activated using a kernel module provided by the project `omke`. The sources are available from `http://sourceforge.net/projects/omke/`. Once the kernel module is loaded, the missing buttons are working. Like the Acer driver, normal scancodes are generated and no additional software is needed. A list of supported laptops by this driver is found in Appendix C, §C.4.

Chapter 3

Requirements Specification

The requirements specification is the basis upon which the design and architecture, as well as the implementation of the Input Abstraction Layer are built. The relevant kernel drivers' interfaces are summed up and described in a more abstract way than they were discussed in the previous chapter. Moreover, the attributes of the Input Abstraction Layer's output interface are described. The functional requirements describe the essential functions and features necessary to eliminate the existing gap between the available kernel drivers and the user interface. The current issues regarding Linux input are discussed, solutions are addressed. The performance and quality requirements describe the goals which have to be achieved.

It is important to bear in mind that the Input Abstraction Layer is a daemon process. Once running, the Input Abstraction Layer is an autonomous background process which does not offer a user interface at run time. It is not meant to be controlled by user interaction at run time.

3.1 Required Interfaces

This section sums up the existing input interfaces implemented by the drivers discussed in Chapter 2. Additionally, current issues and their solutions are addressed. Subsequent to this summary, the abstract output interface is described. This interface needs to be implemented by the Input Abstraction Layer.

Input Event Interface

The event interface (Chapter 2, §2.5) is the most important interface regarding Linux input as it is reporting all input events generated by the various input drivers which use the Linux input core—especially the keyboard drivers. The major issue which can be solved by using the event interface is the following: if an input driver is reporting an event to the input core for which no translation from scan- to keycode is known, the input event is not processed by the event handlers (e.g. `keyboard.c`). Thus, the event does not get delivered to TTYs or the X Window System. The delivery of the events to the TTYs or the X Window System is the prerequisite for user space applications to receive the event. However, the event interface handles all events coming from the input core—whether there is a translation for the event's scancode or not. Table 3.1 sums up the attributes of the event interface.

Driver/Interface name:	Event Interface
User space interface:	`/dev/input/event`n (minor 13, major 64+n)
Data type:	binary
Permissions:	read and write access for `root`
Read access:	`open(2)`, `read(2)` and `ioctl(2)`
Write access:	`ioctl(2)`

Table 3.1: Event Interface Attributes

User space interface:	`/proc/acpi/event`
Driver/Interface name:	ACPI Event Interface
Data type:	string
Permissions:	read access for `root`
Access method:	`open(2)` and `read(2)`
User space interface:	`/var/run/acpid.socket`
Driver/Interface name:	ACPI Daemon Socket
Data type:	string
Permissions:	read and write access for all users (only read access is used)
Access method:	`socket(2)` (Unix domain socket)

Table 3.2: ACPI Event Interface Attributes

ACPI Event Interface

The ACPI event interface reports all ACPI events—events triggered by the ACPI button driver as well as events triggered by other ACPI drivers. User space applications can only access the interface `/proc/acpi/event` if they have `root` privileges. This would not be a serious issue if the only input events reported using ACPI would be the events of the power button, sleep button and lid switch. But as described in Chapter 2 (§2.6), ACPI is also used to report function key events. These function key events are interesting for user space applications which do not run with `root` privileges, too. For example, a function key with an imprint for *change volume* or *start mail program* should be accessible by all users—not only to `root`.

If the previously addressed ACPI daemon `acpid` is running, user space applications without `root` privileges are able to receive the ACPI events by connecting to the Unix domain socket `/var/run/acpid.socket`. The daemon `acpid` is sending all occurring ACPI events to the connected process or processes. No `root` privileges are required to connect and read from the socket. It is up to the process reading from the socket to pick out the interesting events.

The attributes of both the ACPI event interface and the ACPI daemon are summarized in Table 3.2.

Specific Event Interfaces

Several interfaces for user input are only available on specific systems. But this fact does not make these interfaces less important. All drivers for these specific interfaces are implementing their own interface rather than using the Linux input core or ACPI event interface. Thus, they depend on user space applications using the specific interface to process input events. Common issues are that there is no reasonable user space application available or users have

User space interface:	`/dev/i8k` (minor 10, major 144)
Driver/Interface name:	Dell SMM Driver
Data type:	string
Permissions:	read access for all users
Access method:	`open(2)` and `read(2)`
User space interface:	`/dev/nvram` (minor 10, major 144)
Driver/Interface name:	IBM ThinkPad Button (NVRAM Driver)
Data type:	binary
Permissions:	read and write access for `root`
Access method:	`open(2)` and `read(2)`
User space interface:	`/dev/sonypi`(minor dynamic, major 10)
Driver/Interface name:	Sony Programmable I/O Control Device Driver
Data type:	binary
Permissions:	read access for `root`
Access method:	`ioctl(2)`
User space interface:	`/proc/acpi/toshiba/keys`
Driver/Interface name:	Toshiba ACPI Driver
Data type:	string
Permissions:	read and write access for `root`
Access method:	`open(2)`, `read(2)` and `write(2)`
User space interface:	`/proc/toshiba`
Driver/Interface name:	Toshiba SMM Driver
Data type:	string
Permissions:	read access for all users
Access method:	`ioctl(2)`

Table 3.3: Specific Event Interface Attributes

trouble to set up the driver and the corresponding application. The attributes of the specific event interfaces are summarized in Table 3.3.

Output Interface

The output interface has to be implemented. It does not yet exist. All events which can be gathered using the above mentioned input interfaces have to be reported to this output interface. All user space applications should be allowed to use the output interface to receive input events. The events delivered by the output interface should have a common representation no matter which input interface they originally came from. The common representation ensures that the applications receiving events from the output interface are able to handle the events even if the input interfaces are changing.

The output interface is the most important part of the Input Abstraction Layer. By means of this interface, the mentioned gap between the existing Linux input device drivers and the user interface can be closed. The attributes of the interface are elaborated in Chapter 4, §4.4. Chapter 5 discusses the output interface's implementation in detail.

3.2 Functional Requirements

The functional requirements are derived from the current problems regarding user input.

Unattended Input Events

The major problem is that many events received by the Linux kernel actually are not arriving at to user space applications. Thus, they can not be used. The first functional requirement is the implementation of a user space interface which is reporting to up until to now unattended events.

Until now, each keypress event which does not have a translation from its scancode to a keycode is recognized by the Linux input core, but the event does not reach user space applications as the event interface is not utilized to its full extent. Even though ACPI events are accessible by either using the ACPI event interface or the ACPI daemon's socket, applications do not make full use of these interfaces. This applies to events of the above mentioned specific input device drivers, too: input events are delivered to user space interfaces which are not utilized by applications.

Elimination of Barriers

The aforementioned input interfaces have different access permissions—some are accessible by all users, some only by users with special permissions (`root`). No special permissions shall be necessary in order to access the user space interface offered by the Input Abstraction Layer. All user space processes should be able to receive events by listening on the interface. This second functional requirement implies a security threat, which is addressed in Section §3.5.

Unified Data Representation

The third functional requirement applies to the representation of the events reported by the Input Abstraction Layer's output interface. To avoid inconsistency, the events should have a common data representation, independent of the actual event source.

Permanent Interface

The output interface itself should not change its location. No matter how it is implemented, it should always be accessible at the same location. This ensures that applications can easily check whether the Input Abstraction Layer is available or not. The permanent interface is the fourth functional requirement.

Independent Interface

The fifth and last functional requirement states that the Input Abstraction Layer should be accessible to all applications. This time, access does not apply on the permissions as described by functional requirement number two. Applications should be able to use the Input Abstraction Layer's interface independent of the programming language they are implemented in.

3.3 Performance Requirements

As mentioned in the introduction of this chapter, the Input Abstraction Layer is an autonomous background process which is always running. No matter whether there are any input events to be processed or not, the daemon has always to be poised, to work them up as soon as they occur. For this reason, it is essential that the Input Abstraction Layer does not consume more system resources than needed—this applies to both processor usage and memory usage.

To reduce the consumption of system resources, user space applications should not need to poll in order to access the Input Abstraction Layer's output interface. An application using the output interface should receive a signal from the Input Abstraction Layer once an input event occurs. On the one hand, this ensures that input events arrive at user space applications with the slightest possible delay. On the other hand, if applications do not need to poll, the Input Abstraction Layer's output interface induces less consumption of system resources.

For any user input, the peak response time is crucial. Normal keyboard input is acceptable if the delay between keypress and response is within 25 milliseconds. The Input Abstraction Layer has to meet requirements regarding response times as well. Depending on the input interface, the peak response time varies since some of them need to be polled. In general, the time between the user action and the arrival of the input event at the user space application should not exceed a limit of 25 milliseconds. Since the Linux kernel does not meet real-time requirements, the response time can not be guaranteed.

3.4 Quality Attributes

The Input Abstraction Layer has to fulfill the quality attributes which should be fulfilled by all daemon processes. The daemon has to run stable and without interruption under normal conditions. In case of a failure the daemon should handle the exception and issue an expressive error message.

The configuration should be carried out without much hassle. Critical parts should be emphasized in the documentation. To address the many different system configurations the Input Abstraction Layer should run on, the daemon has to be highly customizable to the different needs.

The base configuration of the daemon should cover all available input interfaces. This eliminates troubles for users with little experience. On start-up, the daemon has to identify the available input interfaces and verify that it can access them accordingly.

To be prepared for future input interfaces, the daemon has to be easily expandable. This also ensures that other developers are willing to enhance the Input Abstraction Layer by adding support for not yet supported input interfaces.

3.5 Security Attributes

A sequence of several input events can contain sensitive data, e.g. passwords. On that score, the Input Event Layer should report all but alpha numeric input events. Otherwise, a serious security impact would be raised: since all user processes shall be able to receive input events (fifth functional requirement, §3.2), a process could abuse the Input Abstraction Layer to gather sensitive data.

Chapter 4

Design and Architecture

This chapter describes the Input Abstraction Layer's design and architecture. It connects the requirements specification (Chapter 3) with the actual implementation discussed in Chapter 5. Throughout this chapter, possible implementations are referenced and evaluated.

Section §4.1 discusses the generic event data flow which is common for all input events. The section shows the present problem of unattended input interfaces and points out how this problem can be solved by the Input Abstraction Layer. The remaining sections of this chapter outline the Input Abstraction Layer's structure and its components. Section §4.2 discusses the daemon as core component of the Input Abstraction Layer. Section §4.3 and Section §4.4 describe the Input Abstraction Layer's interfaces.

4.1 Event Data Flow

From an abstract point of view, all input events have a common data flow. In the first place, the user has to press a button on an input device. This user action triggers a hardware event. If a suitable driver for the input device is loaded, the input event gets processed. At this point, the hardware event is merged into a software event. The driver makes this software event accessible to applications either by providing or using a user space interface.

There are two critical paths in this scenario: first, if there is no driver for an input device, its hardware event does not come through. No software event is triggered. Second, if there is no user space process reading from the driver's interface, the software events are dismissed.

Figure 4.1 shows the three possible ways for input events triggered by a user action. The user space application (shown as "Application") receives the input event if there is a driver for the input device and if the application either is reading on the driver's user space interface or gets informed by an event handler (Figure 4.1a). If there is no driver available (Figure 4.1b) or if the user space interface is unattended (Figure 4.1c) the application does not receive the input event. For these two paths, the interruption of the data flow is indicated by the crosses as shown in Figure 4.1a and Figure 4.1b.

All unattended interfaces can be made usable with the Input Abstraction Layer. In case of missing drivers the Input Abstraction layer is futile. Figure 4.2 shows how the Input Abstraction Layer ("IAL") fits into the common event data flow. While the Input Abstraction Layer is reading input events from the input interfaces discussed in Chapter 2, other applications can receive these events by reading a single interface provided by Input Abstraction Layer. Applications which do not directly use the Input Abstraction Layer's event interface can be

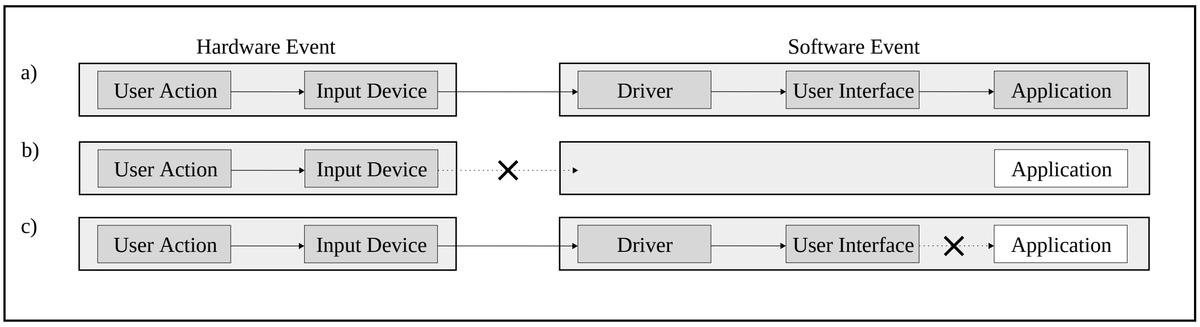


Figure 4.1: Common Event Data Flow

notified about input events, too. This case is shown in Figure 4.2 by means of an interconnect process (“Process”) which reads from the Input Abstraction Layer’s output interface and informs other applications about occurring input events.

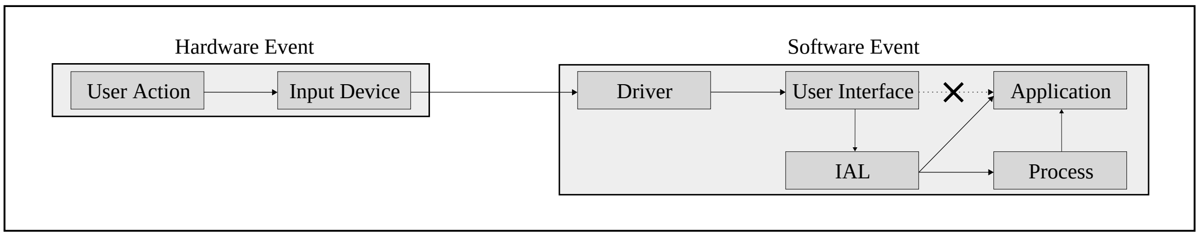


Figure 4.2: Common Event Data Flow with Input Abstraction Layer

4.2 Daemon

The daemon which realizes the Input Abstraction Layer has to provide several features common for system daemons:

- Configuration File Parser
- Command Line Interface (CLI)
- Debug Output

The design and architecture of the modular input interface and the abstract output interface are also part of the daemon, but they are discussed in separate sections to emphasize their importance.

As defined by the Filesystem Hierarchy Standard [Fsh04], host-specific system configuration files should be stored in the directory `/etc/`. The format of configuration files is not determined and hence can be chosen as desired. The configuration files are parsed at start-up of an application—in this case by the Input Abstraction Layer daemon. Well known examples for thoughtful configuration file formats are the Apache HTTP Server and Samba. The Apache HTTP Server configuration file contains so called directives. In the following example, the directive is `Directory` with the argument `"/usr/local/htdocs"`. The paled options

`AllowOverride` and `Order` are options for the directive `Directory` with their corresponding values.

```
<Directory "/usr/local/htdocs">
     AllowOverride All
     Order deny, allow
</Directory>
```

Samba uses a configuration file format which was derived from the `.ini` files used by Microsoft Windows 3.1. This format is divided into sections. Each section has one or more options with a corresponding value. The following example shows the sections `[global]` and `[share1]`. Two options (`workgroup` and `netbios name`) are defined for the section `[global]`, for section `share1`, only one option (`path`) is supplied.

```
[global]
workgroup = WORKGROUPNAME
netbios name = NETBIOSNAME
[share1]
path = /home
```

The parsing of the configuration file can be shifted by using an external library, such as the XML C parser Libxml2 (`libxml2`). Libxml2 implements a decent number of standards like XML, XPath, HTML4 and others. A complete list can be obtained on the project home page located at `http://www.xmlsoft.org/`. Furthermore, Libxml2 is available for many platforms. A notional example for a configuration file format using XML would be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration>
    <!-- node with numeric value -->
    <num_node>1</num_node>
    <!-- node with string value -->
    <alpha_node>string</alpha_node>
    <!-- complex node with attribute and children nodes -->
    <complex_node attribute=attribute_value>
        <node1>yes</node1>
        <node2>10</node2>
    </complex_node>
</configuration>
```

The second approach to configure the Input Abstraction Layer daemon are command line options. It is common to use the command line parser `getopt(3)` to implement the command line interface. Since the options defined in the configuration file interleave with options passed by the command line, it is necessary to specify which source for the daemon options is more significant. Command line options are convenient for setting specific daemon options temporarily—e.g. to test a feature or to change the debug level. Based on this fact, it is sensible to determine the sequence for the evaluation of options:

First, the configuration file is parsed and the included options are set. Secondly, the command line arguments are evaluated and the included options are set. Options previously set by parsing the configuration file are overwritten.

In order to examine the runtime behavior of the Input Abstraction Layer, a multi level logging system is needed. Depending on which logging level is set, the daemon issues more or less debug output. The following logging levels are needed: error (critical errors only), warning (critical errors and warnings), info (critical errors, warnings and information) and debug (all debug messages).

4.3 Modular Input Interface

As demanded by the first functional requirement, the Input Abstraction Layer has to take care of the unattended input interfaces. Depending on the used hardware, input interfaces vary a lot. It is suggestive to design a modular input interface for the Input Abstraction Layer. For each input interface provided by the Linux kernel, a corresponding module for the Input Abstraction Layer is necessary. Such a module is autonomous to a large extent: once the daemon has loaded a module, the module accesses its kernel input interface and reports occurring events.

The Input Abstraction Layer's daemon is responsible for loading the modules upon start-up. Each module needs to implement an initialization routine, which is setting up the access to the kernel interface the module is designed for. If the initialization succeeds, the module subsequently reports input events. Like the daemon, the modules need to be configured. Since the modules are loaded by the daemon, the configuration of the modules needs to be integrated into the daemon functions which implement the parsing of both the configuration file and the command line options.

The daemon, in conjunction with its modules, can be implemented in two different ways: either by using threads or by using an event loop. Programming with threads is difficult and tough to debug. Threads are hard to synchronize and the risk of deadlocks is given. In contrast to an event loop, threads are providing concurrency—several execution streams are running in the same context. An event loop is a single execution stream which is event driven. Occurring events are processed by assigned event handlers. The only significant benefit offered by threads is concurrency. Since the modules do not need to execute code concurrently, it gives no sense to use threads for the implementation. Using threads would actually cause problems, since events would have to be sequenced to ensure that they are reported in the correct chronological order.

To implement the event loop, the GLib (`http://www.gtk.org`) library can be used. GLib is the core library used by the Gimp Toolkit (GTK+) and Gnome, though it has nothing to do with GUI programming or the X Window System. Beside other useful functions, the library offers an excellent way to implement event loops.

It is important to bear in mind that the modules are autonomous regarding what they are doing. But all modules are running in the context of the daemon. This is why the modules are not autonomous regarding when they are actually receiving control to execute their code. To achieve a flexible architecture, adding and removing modules of the Input Abstraction Layer should not require a recompilation of the daemon. This approach can be achieved by using dynamically loaded shared libraries.

Figure 4.3 shows the modular input interface. Incoming input events are shown as dotted arrows on the left of the figure. Input events are reported to the various kernel input device drivers that provide a user space interface (shown as “Input Interface”). Each module (“IAL Module”) is responsible for one input interface. The execution of the module is controlled by the event loop of the Input Abstraction Layer. The modules report occurring events to the output interface using a function provided by the daemon. The dotted arrows on the right indicate the subsequent event flow to user space applications, which are using the Input Abstraction Layer’s output interface.

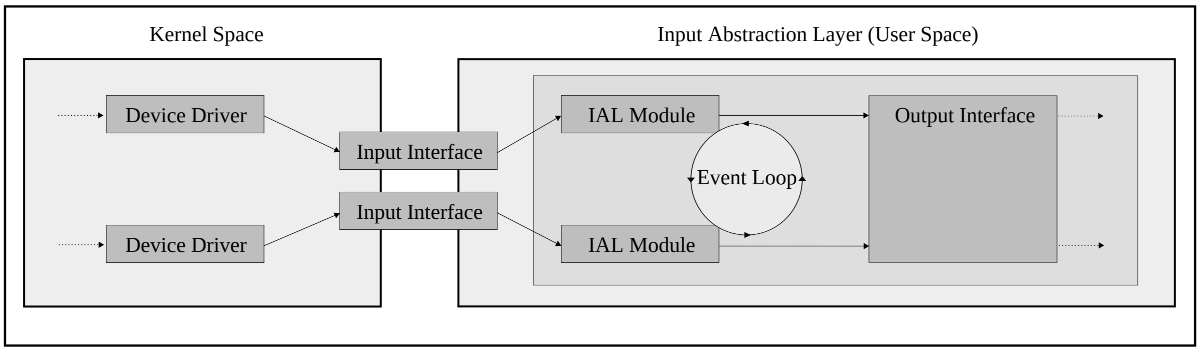


Figure 4.3: Modular Input Interface

4.4 Abstract Output Interface

Input events received by the modules are reported using the daemon’s output interface. The actual abstraction is done by the modules—not by the daemon. A module translates concrete events read from the kernel drivers’ interface to abstract events. These abstract events are sent to the abstract output interface provided by the daemon. This interface is accessible to all user space processes which want to receive input events, as demanded by the second functional requirement. This implies the need for an interprocess communication (IPC) between the daemon and user space applications.

Linux provides a number of different methods for interprocess communication: signals, named and unnamed pipes, message queues, semaphores, shared memory and sockets. The two major Linux desktop environments for the X Window System—Gnome and KDE—have their own IPC frameworks. Gnome is using ORBit, a CORBA 2.4-compliant Object Request Broker. KDE is using Desktop Communications Protocol (DCOP) for IPC. DCOP depends on the Qt framework. Both, ORBit and DCOP are built upon existing IPC mechanisms to offer a better and more comfortable API to user space programs, which want to communicate with each other. Another approach for interprocess communication is the message bus system D-BUS which was recently introduced. D-BUS can neither be assigned to Gnome nor to KDE.

In contrast to ORBit and DCOP, D-BUS merely depends on an XML parser, such as the aforementioned `libxml2`. This parser is required by D-BUS to parse its configuration files. D-BUS was designed to offer a low latency and low overhead IPC protocol for any application—with no focus on a specific desktop environment or programming language. The D-BUS protocol is a binary protocol and therefore, messages do not have to be serialized. This benefits the efficiency of D-BUS. Freedesktop.org (`http://www.freedesktop.org/`) serves as the project lead. This emphasizes the will of implementing an independent IPC framework, that does not displease any Gnome or KDE developer, as Freedesktop.org is working on projects supporting the collaboration and interoperability of desktop environments.

Beside the small amount of dependencies, D-BUS distinguishes itself by offering bindings for many high-level programming languages. This fact satisfies the fifth functional requirement (Chapter 3, §3.2), which states the independence regarding the programming language used by any application utilizing the Input Abstraction Layer's output interface.

D-BUS offers two different types of message buses: the system bus and the session bus. The system bus is used by system daemons and D-BUS itself, whereas the session bus is used for user space applications. While there is only one system bus, it is possible to create an arbitrary number of session buses. By default, the configuration for the system bus is secured to prevent user space applications from receiving or sending messages to the interfaces of the bus. While the system bus is started with the start-up of D-BUS (e.g. system boot), session buses are started when needed. D-BUS provides several message types to realize the interprocess communication between applications using one of the buses: method calls, method returns, signals and errors. Messages can have arbitrary arguments. The data types for these arguments are defined by the D-BUS protocol and cover basic as well as complex data types.

The interprocess communication is transparent for both the Input Abstraction Layer and the application using its output interface. Figure 4.4 shows the sequence of the data flow. First, modules report input events by calling a function provided by the daemon. The abstraction is realized at this point: the module's input is concrete, its output is abstract. The modules use a common function provided by the daemon to report events to the output interface. This function invokes an IPC function for sending the abstract input event. User space applications receiving events from the abstract output interface are using a corresponding receive function offered by the utilized IPC. It is also possible that applications which do not directly read from the Input Abstraction Layer's output interface make profit out of the Input Abstraction Layer. This case requires a broker process (shown as "Process"), which receives events from the daemon and relays them to an input interface the user space application is using.

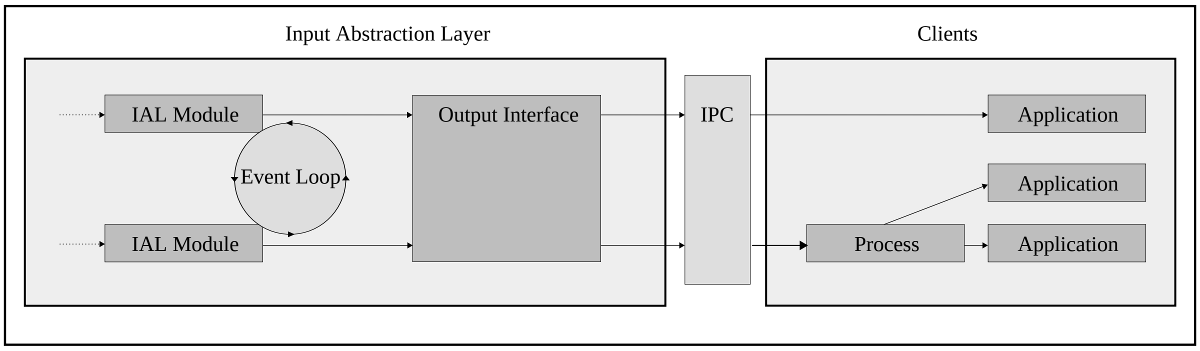


Figure 4.4: Abstract Output Interface

Chapter 5

Implementation

This chapter describes the implementation of the Input Abstraction Layer in detail. The actual process of the implementation was preceded by a comprehensive code review of miscellaneous open source projects. The review gave informative hints on how the Input Abstraction Layer's requirements could be realized. Resulting from this, several parts of the Input Abstraction Layer's implementation are derived from the reviewed source code.

The mentioned path names of the source code discussed in this chapter are relative to the root of the Input Abstraction Layer's source tree (Appendix D, §D.2). The three main components of the Input Abstraction Layer are termed "daemon", "library" and "module". An overview of the components is outlined in Figure 5.1. It shows the three core components and their static context. The daemon and the modules are both using functions provided by the library. External libraries used by the components are not indicated. Common data structures used by the daemon and the modules are part of the library. Among others, the library implements functions used for the interprocess communication with other processes.

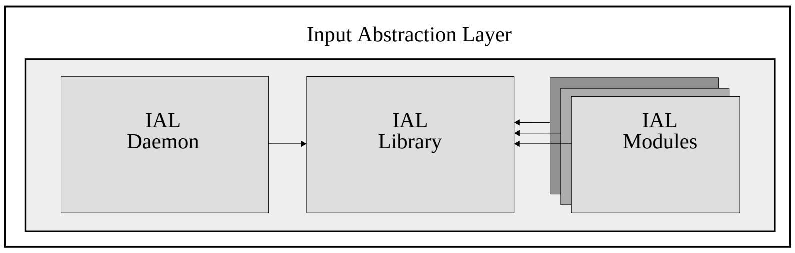


Figure 5.1: Input Abstraction Layer Components

Firstly, common static data structures of the Input Abstraction Layer's components are discussed. Comparable to a physical engine, it is not possible to achieve a complete understanding of the inner operations simply by observing the engine running. A static view of the engine's components and the environment it runs in is the prerequisite for examining the order of events afterwards. Thus, knowing the data structures is essential for the understanding of the component's functions and the dynamic cooperation of the components at run time. All components are implemented in C. Functions of the GNU C Library (`glibc`, `http://www.gnu.org/software/libc/`) are not discussed in detail. Other libraries used for the implementation are referred and elaborated in detail where needed.

The subsequent sections of this chapter separately discuss the implementation of the three components. Important attributes of the components are emphasized and the dynamic aspects are taken into account, too. In the end of this chapter several modules are discussed which impart the actual functionality of the Input Abstraction Layer.

The complete source code is documented using Doxygen (`http://www.doxygen.org/`). Doxygen is an open source documentation system for various programming languages. The Doxygen documentation addresses developers interested in the Input Abstraction Layer's implementation while this chapter describes the implementation in a legible way to address non-developers as well. Further details about the Doxygen documentation can be found in Chapter 7, §7.2.

5.1 Common Data Structures

The Input Abstraction Layer's components are tied together by using common static data structures. These data structures are defined in the header files of the library found in the directory `libial`. These header files are subdivided into three separate files: `libial.h`, `libial_mod.h` and `libial_log.h`. The main header file `libial.h` includes the remaining two header files. Beside this, `libial.h` also includes header files of external libraries used by all components: D-BUS and GLib. Apart from its similar name, the library GLib has nothing in common with the glibc GNU C Library. Figure 5.2 shows the include dependency graph for `libial.h`.

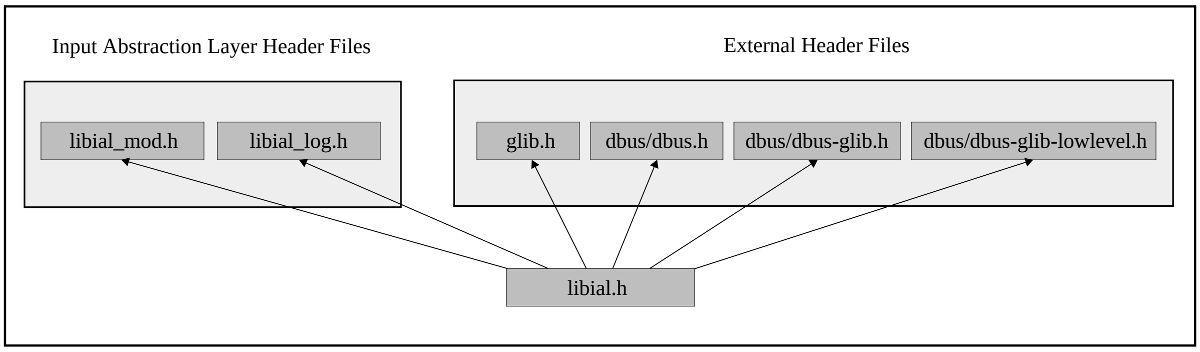


Figure 5.2: Include Dependency Graph for `libial.h`

Abstract input events are represented using the data structure `IalEvent`. This data structure is used by two components of the Input Abstraction Layer—the library and the modules. The structure is declared in `libial.h`:

```
typedef struct IalEvent_s {
     const char *sender;
     const char *source;
     const char *name;
     int raw;
} IalEvent;
```

Modules use the structure `IalEvent` in conjunction with a wrapper function to report abstract input events. The library `libial` implements this wrapper function and sends the event using D-BUS. The meaning of the structure's members is discussed in Section §5.2 and §5.3.

All modules are based on two static data structures: `ModuleData` and `ModuleOption`. The main structure `ModuleData` represents the common ground on which every module for the Input Abstraction Layer is based on. The structure `ModuleOption` is required to represent a module's configuration option. Modules usually have a set of configuration options represented as an array of `ModuleOption` structures. Apart from the modules, the daemon utilizes the two structures for initialization, configuration and controlling the modules.

The use of both structures, their general importance at runtime and the meaning of the structures' members are discussed in Section §5.3 and §5.4.

5.2 Library

The library `libial` provides functions used by the remaining two components—the daemon and the modules—and external applications. It implements wrapper functions for the interprocess communication and the logging system. Hence, the library is split into two parts: `libial.c` and `libial_log.c`. The daemon as well as the modules have to be linked against `libial` to utilize the library's functions. Additionally, the library implements a function which can be used by external applications to receive events from the Input Abstraction Layer. Due to the fact that the library wraps most of the interprocess communication, this section first describes the implementation of the interprocess communication using D-BUS.

D-BUS

In order to use D-BUS for interprocess communication, it first has to be properly set up. D-BUS is implemented as a user space process (D-BUS daemon, `dbus-daemon-1(1)`) which provides the two different message buses described in Chapter 4, §4.4. Since the Input Abstraction Layer runs as a system daemon, it uses the system message bus. By default, this message bus is well protected to prevent unauthorized processes from using it. The configuration of the system message bus is stored in the file `/etc/dbus-1/system.conf`. It follows a deny policy. This means, that by default, all operations on the system bus are denied for everyone. The following excerpt from `system.conf` shows the corresponding implementation:

```
<policy context="default">
    <deny send_interface="*"/>
    <deny receive_interface="*"/>
    <deny own="*"/>
    <allow user="*"/>
    <includedir>system.d</includedir>
</policy>
```

The first three tags encompass the default deny policy. On start-up the D-BUS daemon sequentially parses the configuration file and hence denies send and receive operations to any interface of the system message bus. The tag `<deny own="*"/>` further prevents processes from creating services on the system message bus. The following tag `<allow user="*"/>` is the first purposed hole in the deny policy: it allows all processes to connect to the system bus. Further, the D-BUS daemon parses additional configuration files located in the directory `/etc/dbus-1/system.d/`. System daemons utilizing D-BUS for interprocess communication store their respective D-BUS configuration in this directory.

A D-BUS service is the basic requirement for interprocess communication using D-BUS. Arbitrary D-BUS interfaces can be applied to a D-BUS service once it is created. Sending and receiving messages on a D-BUS interface can be allowed or denied for certain users. The Input Abstraction Layer creates the service `com.novell.Ial` and the D-BUS interface (`com.novell.Ial.Event`) in order to implement the abstract output interface. The Input Abstraction Layer's D-BUS configuration file `/etc/dbus-1/system.d/ial.conf` contains the following data:

```
<policy context="default">
    <deny own="com.novell.Ial"/>
    <deny send_interface="com.novell.Ial.Event"/>
    <allow receive_interface="com.novell.Ial.Event"/>
</policy>

<policy user="root">
    <allow own="com.novell.Ial"/>
    <allow send_interface="com.novell.Ial.Event"/>
    <allow receive_interface="com.novell.Ial.Event"/>
</policy>
```

First, the policy applies to processes owned by all users but `root`: these processes are not allowed to own the service `com.novell.Ial` and they are not allowed to send to the interface `com.novell.Ial.Event`. However, they are allowed to receive messages from the interface. Secondly, a policy for processes owned by the user `root` is defined: these processes are allowed to create the service and they are allowed to send and receive from the event interface.

Starting the D-BUS daemon does not invoke the creation of the service or the interface. The D-BUS configuration only describes the permissions applied to services and interfaces. In the case of the Input Abstraction Layer, both the service and the interface are created at the daemon's start (Section §5.4).

Programming Interface

The Input Abstraction Layer's library provides three wrapper functions for the interprocess communication: `ial_dbus_connect()`, `event_send()` and `event_receive()`. The daemon utilizes `ial_dbus_connect()` to establish a connection to the D-BUS system message bus. The function `event_send()` is used by the Input Abstraction Layer's modules to send input events to the abstract output interface. External applications can simplify the receiving of abstract input events by using the library's function `event_receive()`.

The connection to the D-BUS system message bus is represented as a global variable of the daemon. Once established, it is used for all further interprocess communication. The function `ial_dbus_connect()` is defined in `libial.c`. The following source code is an excerpt of this function:

```
/* Check whether D-BUS connection is already established. */
if (dbus_connection == NULL) {
    /* Connect to the D-BUS system message bus. */
    dbus_connection = dbus_bus_get(DBUS_BUS_SYSTEM, &dbus_error);
}
```

The type of the global variable `dbus_connection` is `DBusConnection`. It represents a D-BUS connection. The D-BUS function `dbus_get_bus()` establishes the connection and subsequently returns the established connection. The first argument (`DBUS_BUS_SYSTEM`) states that a connection to the system message bus is requested. Once issued, the D-BUS daemon checks if the request is allowed by comparing the request with the policy defined by the D-BUS configuration.

The modules report input events calling the library's function `event_send()`. This function requires a pointer to an `IalEvent` structure—representing the abstract input event—as an argument:

```
void event_send(IalEvent *);
```

The `IalEvent` structure has to be set up by the module which is calling `event_send()` (§5.3). Upon being called, `event_send()` creates a D-BUS message and sends this message to the Input Abstraction Layer's D-BUS interface. This is done by appending each member of the referred `IalEvent` structure to a D-BUS message. The following source code is a modified and shortened excerpt of `event_send()`, supplemented with the explanatory comments.

```
/* Create D-BUS message and message iterator. */
DBusMessage *dbus_message;
DBusMessageIter dbus_it;

/* Create a new message. */
dbus_message = dbus_message_new_signal("/com/novell/Ial/Event",
                                       "com.novell.Ial.Event",
                                       "event");

/* Initialize the message iterator. */
dbus_message_iter_init(dbus_message, &dbus_it);

/* Append message arguments. */
dbus_message_iter_append_string(&dbus_it, event->sender);
dbus_message_iter_append_string(&dbus_it, event->source);
dbus_message_iter_append_string(&dbus_it, event->name);
dbus_message_iter_append_int32(&dbus_it, event->raw);

/* Send the message. */
dbus_connection_send(dbus_connection, dbus_message, NULL);
```

`DBusMessage` is the data type representing a D-BUS message. The D-BUS data type `DBusMessageIter` is an iterator representing the arguments of a `DBusMessage`. The function `dbus_message_new_signal()` creates a new `DBusMessage` of type `SIGNAL` and returns a pointer to the created structure. The arguments specify the message path (`/com/novell/Ial/Event`), the D-BUS interface (`com.novell.Ial.Event`) and the signal name (`event`). These values are essential for the D-BUS daemon to process the D-BUS message.

The message type `SIGNAL` is chosen since it matches the kind of an event. It is a unidirectional message sent by one process and received by an arbitrary number of processes. Other message types provided by D-BUS are `METHOD_CALL` and `METHOD_RETURN`. These message types can be used to realize remote procedure calls with D-BUS.

D-BUS provides several functions to append arguments to a `DBusMessageIter` in particular, for each of its basic data type. In this case, the basic data types `string` and `int32` are used. Since the Input Abstraction Layer is implemented using C, `char *` is mapped to the basic D-BUS data type `string`, `int` is mapped to the basic D-BUS data type `int32`. An application receiving the D-BUS message might be implemented using another programming language, thus a corresponding type conversion has to be performed by D-BUS upon reception. After appending all members to the D-BUS message, it is sent to the message bus.

All messages sent by the Input Abstraction Layer's modules can be received by any application. This is ensured by the described policy for the D-BUS daemon (allow all processes to connect to the system message bus) in combination with the policy for the Input Abstraction Layer (allow all processes to receive from the interface `com.novell.Ial.Event`). An application which wants to receive messages has to set up a connection to the D-BUS system message bus. Using this connection, the application has to install a so called *match* to receive abstract input events. A match can be compared to a filter mechanism: only messages matching the rules of the match will get through. The following code shows how a match is created:

```
/* Add a message filter to the D-BUS connection. */
dbus_connection_add_filter(dbus_connection, event_callback, NULL, NULL);

/* Add a match to the D-BUS connection. */
dbus_bus_add_match(dbus_connection,
                   "type='signal',
                    interface='com.novell.Ial.Event',
                    path='/com/novell/Ial/Event'",
                   &dbus_error);
```

The function `dbus_connection_add_filter()` adds a message filter to the established D-BUS connection. This filter defines a message handler. In this case, the message handler is called `event_callback()`. The third and fourth arguments of `dbus_connection_add_filter()` are optional and set to *NULL*. After creating the filter, a match has to be added. The function `dbus_bus_add_match()` is used for this purpose. The second argument of the function call is the rule applied by the match. In this case, the rule states that messages of the type `signal` on the D-BUS interface `com.novell.Ial.Event` using the path `/com/novell/Ial/Event` are selected. After applying this match, the callback function `event_callback()` is invoked every time a message is sent on the D-BUS matching this rule. Beside others, D-BUS invokes the callback function with an argument (`DBusMessage *`) which references the message. The following source code is an example implementation of `event_callback()`:

```
void event_callback(DBusConnection *connection, DBusMessage *dbus_message,
                    void *user_data)
{
    /* Create abstract input event. */
    IalEvent event;

    /* Read arguments from D-BUS message. */
    dbus_message_get_args(dbus_message, &dbus_error,
                          DBUS_TYPE_STRING, &event.sender,
                          DBUS_TYPE_STRING, &event.source,
                          DBUS_TYPE_STRING, &event.name,
                          DBUS_TYPE_INT32, &event.raw,
                          DBUS_TYPE_INVALID);
}
```

The abstract input event is represented by `event` using the `IalEvent` structure. Each argument of the message `dbus_message` is defined by two fields: the first field defines the D-BUS data type of the argument, the second field defines where the value of the argument should be stored. In this case, the message arguments are copied to the corresponding members of `event`. The last argument, `DBUS_TYPE_INVALID`, states that no further arguments of the D-BUS message are expected. The library `libial` provides a wrapper function called `event_receive()` which relieves the receiving of Input Abstraction Layer messages. Applications which want to utilize this function have to be linked against `libial`.

Recapitulating, Figure 5.3 shows the interprocess communication between the Input Abstraction Layer and applications. Abstract input events are sent as messages to the D-BUS interface `com.novell.Ial.Event` by the modules. Applications can receive these messages either by using the `event_receive()` function provided by the library `libial`, or by accessing the D-BUS interface directly.

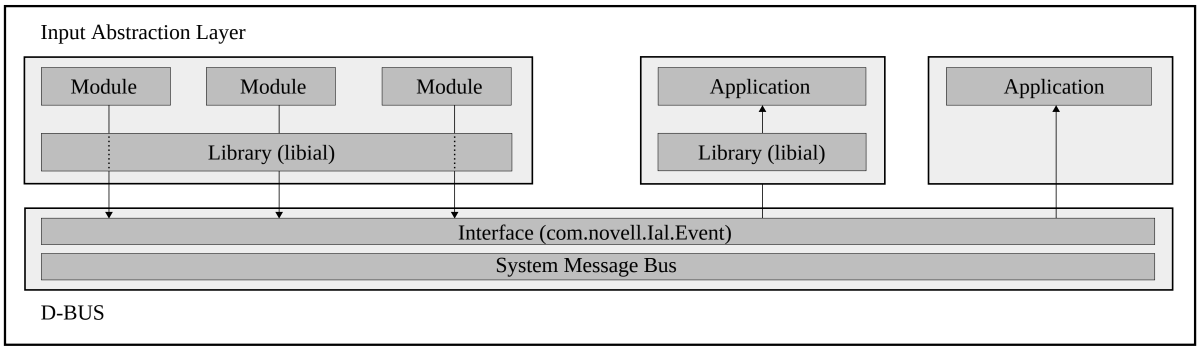


Figure 5.3: Input Abstraction Layer Using D-BUS

Logging System

Apart from providing the wrapper functions for the interprocess communication, the library implements a logging system which is used by both the daemon and the modules. The code of the log system was originally based on code from the HAL project (`http://hal.freedesktop.`

`org`) which is released under the terms of the Academic Free License version 2.0. The logging system is implemented by `libial_log.c` and `libial_log.h`. It provides four different priorities: `LOGPRI_ERROR`, `LOGPRI_WARNING`, `LOGPRI_INFO` and `LOGPRI_DEBUG`—with `LOGPRI_ERROR` being the highest and `LOGPRI_DEBUG` being the lowest priority. The priorities are realized as an enumeration ranging from *0* to *3*:

```
enum {
    LOGPRI_ERROR = 0,
    LOGPRI_WARNING = 1,
    LOGPRI_INFO = 2,
    LOGPRI_DEBUG = 3
};
```

Each priority has a corresponding macro defined in `libial_log.h`. These macros are used by the daemon and the modules to issue log entries. Errors are issued by using the macro `ERROR(`*expr*`)`. Correspondingly, warnings are issued using the macro `WARNING(`*expr*`)`, informative messages are issued using `INFO(`*expr*`)` and debug messages are issued using the macro `DEBUG(`*expr*`)`. All macros are invoked in the same way as `printf(3)`: the format of *expr* is the corresponding format string. The variable `priority` (`libial_log.c`) contains the current log level priority. It is set by the daemon using the library's function `log_level_set()`. Once the log level is set, only log messages less than or equal to the current log level are reported. The lowest log level is `LOGPRI_ERROR` (*0*) which ensures that errors are always reported. The following example shows the detailed application flow of the logging system on the basis of the macro `INFO(`*expr*`)`. The macro is defined as:

```
#define INFO(expr)                                                 \
        do {                                                       \
           log_setup(LOGPRI_INFO, __FILE__, __LINE__, __FUNCTION__);  \
           log_output expr;                                        \
        } while(0)
```

The three other macros are implemented in the exact same way. The only difference is the first argument passed to `log_setup()` which always corresponds to the macro:

- `ERROR(`*expr*`)` calls `log_setup()` with `LOGPRIO_ERROR`
- `WARNING(`*expr*`)` calls `log_setup()` with `LOGPRIO_WARNING`
- `DEBUG(`*expr*`)` calls `log_setup()` with `LOGPRIO_DEBUG`

Granted that the daemon issues the log entry `INFO(("Initialization successful."))` in the function `main()` at line 397 of `iald.c`, the preprocessor dissolves the macro to the following code:

```
do {
   log_setup(LOGPRI_INFO, "iald.c", 397, __FUNCTION__);
   log_output("Initialization successful.");
} while(0);
```

The `do {...} while(0)` loop is run once and encapsulates the invocation of `log_setup()` and `log_output()`. Without this encapsulation, errors could occur if the macro is used in conjunction with an `if-else` construct. The function `log_setup()` stores the values of the function arguments to variables used by the subsequently called function, `log_output()`. The preprocessor can not dissolve `__FUNCTION__` since it is not a macro. The function name `__FUNCTION__` is therefore dissolved at compilation time. The function `log_output()` first checks whether the priority of the issued log entry is less than or equal to `priority`. In case of using the macro `INFO(`*expr*`)` a log entry is reported if `priority` equals `LOGPRIO_INFO` or `LOGPRIO_DEBUG`. It is not reported in case of `priority` being set to either `LOGPRIO_WARNING` or `LOGPRIO_ERROR`. The advantage of the logging system gets more obvious when looking at the corresponding log entry for the log request `INFO(("Initialization successful."))`. The log entry is:

```
Info:    iald.c:397 main(): Initialization successful.
```

The single line of code—`INFO(("Initialization successful."))`—results in a meaningful log entry. The log entry provides information about the source file, the line number and the function name of the original log request. The logging system assists the development of the Input Abstraction Layer in two ways: on the one hand, log requests are brief commands which automatically get enriched with additional information about the context. On the other hand, the logging priority gives a flexible control about which log entries should be reported and which should be suppressed.

5.3 Modules

The Input Abstraction Layer's modules are performing the actual abstraction of the input events. Each module is responsible for one kernel input event interface. Thus, the module knows how the interface is accessed and how its input events are translated to abstract input events (`IalEvent`). All modules have to comply with the following requirements:

– Define an array of `ModuleOption`

– Define a structure of type `ModuleData`

– Define a function `mod_get_data()`

The `ModuleData` structure defines all information needed by the daemon to load the module at run time: it contains module specific information and a pointer to the array of the module options (`ModuleOption`). The daemon invokes the module's function `mod_get_data()` to access the `ModuleData`. It is required that the function `mod_get_data()` returns a pointer to the `ModuleData` structure. This pointer enables the daemon to access all relevant data structures of the module.

All modules are implemented as dynamically loaded shared libraries. The daemon provides a module loader which implements all necessary functions to handle the modules at run time (§5.4).

Data Structure

Both data structures `ModuleData` and `ModuleOption` are declared in `libial_mod.h`. The structure `ModuleData` is slightly shortened compared to the source code.

```
typedef struct ModuleOption_s {
    const char *name;
    char value[MAX_BUF];
    const char *descr;
} ModuleOption;

typedef struct ModuleData_s {
    const char *name;
    const char *token;
    const char *version;
    const char *author;
    const char *descr;
    ModuleOption *options;
    gboolean(*load) (void);
    gboolean(*unload) (void);
} ModuleData;
```

The structure `ModuleOption` represents a module's configuration option. The member `name` contains the name of the option followed by `value` in which the option's value is stored. It has to be initialized statically with a default value. A description of the option is stored in `descr`. As mentioned above, all modules are required to define an array of `ModuleOption` structures. It is essential that the last element of the array is *NULL*. This is necessary since the size of the array varies for each module, as it reflects the number of the module's options. The daemon iterates the `ModuleOption` using pointer arithmetic and stops the iteration as soon as an element with the value *NULL* is found.

The first five members of the structure `ModuleData` are strings providing information about the module. A meaningful name of the module is stored in `name`, `token` contains a minimal name for the module. The member `version` informs about the module's version number, `author` contains the author's name. The module description is stored in `descr`. The member `options` is a pointer to the module's options stored in the already mentioned array of `ModuleOption`. The two members `load()` and `unload()` are function pointers. The module has to implement the two functions accordingly. These functions are responsible for loading and unloading the module, and are invoked by the daemon. The return type of `load()` and `unload()` is `gboolean` which is not a standard data type of the GNU C Library. It is a data type provided by GLib and can either be *TRUE* or *FALSE*. For example, the array `mod_options` of the `ModuleOption` structure, for a module having the two configuration options `disable` and `verbose`, would be implemented by defining:

```
ModuleOption mod_options[] = {
    {"disable", "false", "disable=(true|false)"},
    {"verbose", "true", "verbose=(true|false)"},
    {NULL}
};
```

The following code shows an example of how a module creates the `ModuleData` structure. The member `options` is a pointer to the array `mod_options`.

```
ModuleData mod_data = {
    .name = "Example Input Abstraction Layer Module",
    .token = "example_module",
    .version = "0.1",
    .author = "Timo Hoenig",
    .descr = "This is an example for an Input Abstraction Layer module.",
    .options = mod_options,
    .load = mod_load,
    .unload = mod_unload,
};
```

Figure 5.4 visualizes the main components of a module and the daemon's access to the module's data structures. As discussed, the module is required to define the `ModuleOption` and `ModuleData` structures. Among others, the `ModuleData` structure provides references to the array of `ModuleOption` structures and to the two functions responsible for loading and unloading the module. Thus, the daemon only needs to obtain a pointer to `ModuleData` in order to access all other data of the module. The invocation of `mod_get_data()` returns this pointer to the daemon.

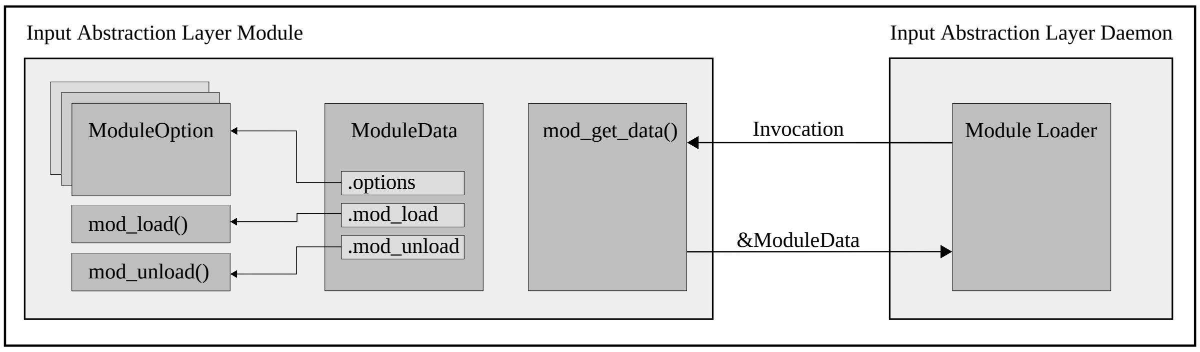


Figure 5.4: Input Abstraction Layer Module

Input Events

Once the modules are initialized, the daemon invokes the loading function `load()` of each module. It is each module's responsibility to verify whether the daemon has set the correct configuration options. A subsequent call to `load()` has to initialize the kernel interface for which the module is designed.

When an input event occurs, the module reports this event by creating an abstract input event `IalEvent` followed by a call to the library's function `event_send()`. The following example illustrates this procedure:

```
/* Create abstract input event. */
IalEvent event;

/* Fill all members. */
event.sender = mod_data.token;
event.source = "Interface Name";
event.name = "Brightness Up";
event.raw = 0x130;

/* Send the event using libial. */
event_send(&event);
```

The member `sender` has to be set to the token of the reporting module. The value of `source` should be set to the name of the original input interface from which the module has received the event. If the module is able to determine a name for the event, the member `name` should be set up. The same applies to the member `raw`: if available, it represents a unique numeric value for the event.

Understanding the full scope of the associations between the modules and the daemon is only possible by investigating the daemon in detail. Hence, the daemon and its run-time behavior are discussed in the next section. Several different module implementations are discussed after the description of the daemon. This will give the final explanation of the context of the modules and the daemon.

5.4 Daemon

The daemon is the Input Abstraction Layer's core component. It sets up the environment by initializing the configuration and loading the modules. It is implemented with the focus of using as few resources as possible. The daemon is subdivided into three parts: the daemon core (`iald.c`), the configuration file parser (`iald_conf.c`) and the module loader (`iald_mod.c`). The daemon itself implements the command line parser and D-BUS functions. The configuration file parser uses `libxml2` to parse the daemon's configuration file `/etc/iald/iald.conf`. The functions provided by the module loader are used to locate, verify, initialize and start the modules. The start-up of the daemon is partitioned into six stages:

– Stage 1: Load modules
– Stage 2: Parse configuration file
– Stage 3: Parse command line options
– Stage 4: Set up D-BUS environment
– Stage 5: Invoke modules
– Stage 6: Enter event loop

Each stage is realized by a specific component of the Input Abstraction Layer daemon. Functions for the first and fifth stage are implemented by the module loader. Stage one performs the detection and verification of available modules. Each valid module is added to a double linked list. Once the modules are detected, stage two is invoked. The configuration file parser is responsible for setting up the configuration for both the daemon and the modules. This applies to the parser of the command line options, too. The command line parser is invoked at the third stage. At this point everything but the D-BUS environment is set up. The daemon establishes a connection to the system message bus and creates the abstract output interface at the fourth stage. Afterwards, the modules are started (stage five) and the daemon enters the event loop (stage six).

Module Loader

The daemon's linchpin is the implementation of the module loader. The module loader provides the functionality to add and remove modules avoiding a recompilation of the whole daemon. This fact makes the Input Abstraction Layer flexible to a high degree. As discussed, the modules are implemented as dynamically loaded shared libraries. The dynamic linking of the modules involves the following problem: at compile time the daemon can not make any assumptions about the modules available at run time. Thus, the daemon needs to detect the modules at start up.

The daemon invokes the detection by calling the function `modules_scan()` of the module loader. This function validates all shared object files (file extension `.so`) in the directory `MODULE_DIR`. The macro `MODULE_DIR` is determined at compile time. It defines the directory where modules of the Input Abstraction Layer are located. By default, `MODULE_DIR` is set to `/usr/lib/ial/modules`. Once `modules_scan()` has found a file with the extension `.so`, the verification routine `module_verify()` gets invoked. As a parameter, the file which has to be verified needs to be passed to `module_verify()`:

```
gboolean module_verify(char *filename);
```

As discussed in Section §5.3, an Input Abstraction Layer module has to define the function `mod_get_data()`. The function `module_verify()` determines if the given file `filename` provides the symbol `mod_get_data` by using the module loader's function `dl_function()`. The two arguments `filename` and `symbol` have to be passed to `dl_function()`:

```
void *dl_function(char *filename, const char *symbol);
```

First, `dl_function()` determines if the file `filename` actually is a shared object file. This check is realized by using `dlopen(3)`. If the given file is not a shared object, `dlopen` returns *NULL*—the verification failed. Otherwise, if the file is a shared object `dlopen` loads the shared object and returns a handle (`void` pointer). This handle is used to examine if the shared object provides the required symbol `mod_get_data`. To accomplish this check, `dlsym(3)` is used. As function arguments, `dlsym` requires the handle provided by `dlopen` and the symbol `symbol` passed to the function `dl_function()`. On success, `dlsym` returns the address where symbol `mod_get_data` is loaded into memory. This address corresponds to a function pointer of the module's function `mod_get_data()`. Figure 5.5 shows the module verification. A module must pass all checks performed by the module loader's functions to be accepted as a valid Input Abstraction Layer module.

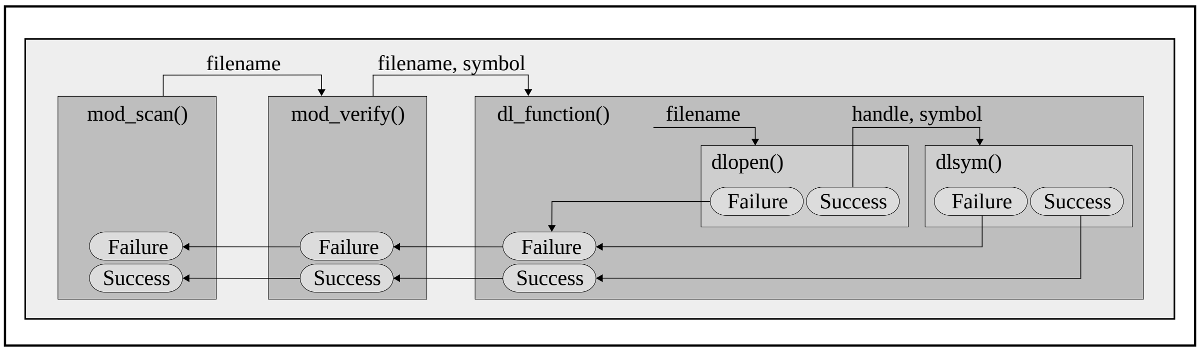


Figure 5.5: Input Abstraction Layer Module Verification

Verified modules are stored in a double linked list. This double linked list is used by the daemon to manage the available modules at run time. An element of the list represents one module and consists of three pointers: a pointer `data` to a structure of type `ModuleData`, a pointer `prev` to the previous element of the list and a pointer `next` to the next element of the list. The data structure for an element of the list is declared in `libial_mod.h`:

```
typedef struct IalModule_s {
   ModuleData *data;
   struct IalModule_s *prev;
   struct IalModule_s *next;
} IalModule;
```

Each verified module is added to the double linked list. To access the double linked list, the daemon defines the global variable `modules_list_head`, which is initialized with the value *NULL*:

```
IalModule *modules_list_head = NULL;
```

Once a verified module is found, the module loader adds it to the list by invoking the function `module_add()`. The function `module_add()` allocates memory for the new element (`IalModule module`) and sets up its members. To obtain the pointer to the module's structure `ModuleData`, the already discussed function `dl_function()` is used. The following source code is an excerpt of `module_add()`:

```
/* Define function returning a pointer to ModuleData. */
ModuleData *(*function) (void);

/* Obtain function pointer. */
function = dl_function(filename, "mod_get_data");
```

Now the function pointer returned by `dl_function()` is assigned to `function`. It can be subsequently invoked, since `function` references the module's function `mod_get_data()`. Once called, it returns the required pointer to the module's data structure `ModuleData`. The returned pointer is assigned to the new list element's `module` member `data`:

```
/* Assign the pointer to the module's ModuleData to the member data. */
module->data = function();
```

The new element is added to the head of the double linked list by setting up the member `prev` and `next` correspondingly:

```
/* Set "prev" to NULL. */
module->prev = NULL;

/* Set "next" to the previous list head. */
module->next = modules_list_head;

/* If the list was not empty before, let the previous list head's member "prev"
 * point to new module.
 */
if (module->next != NULL)
    module->next->prev = module;

/* Let the list head refer the new module. */
modules_list_head = module;
```

Figure 5.6 shows the double linked list with three list elements. The member `data` of each element refers to the corresponding `ModuleData` structure of an Input Abstraction Layer module. The daemon can use this single reference to access all other data structures of the module, such as the module's options `ModuleOption`.

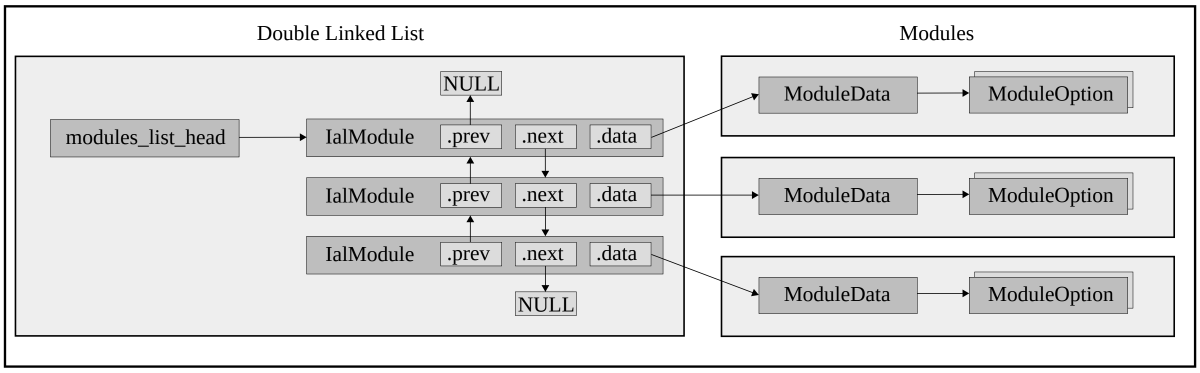


Figure 5.6: Input Abstraction Layer Module List

At this point, the daemon has accomplished stage one—the loading of the modules is accomplished. Though, the modules are not yet running. They are just loaded into memory and the daemon is able to access all necessary data structures. Before actually executing the modules, the daemon needs to set up the configuration for the modules and the daemon itself.

Configuration File Parser

The Input Abstraction Layer's daemon configuration is stored in `/etc/ial/iald.conf`. It is formatted as an XML document. The document root has to be `<ialdconfig>`. Beneath of this document root, all configuration options for both the daemon and the modules are stored. Each child node of the document root is a configuration option. The configuration file parser `conf_parse_file()` evaluates all children nodes using functions of the library `libxml2`.

The configuration options can be divided into two categories: daemon options and module options. The daemon's configuration options are known at compile time. Thus, the parser is able to evaluate them by comparing the configuration file's options with expected options. Since the modules are detected and loaded at run time, their configuration options are not known at compile time. Once the parser has read a configuration option for a module, it has to iterate through the double linked list in order to detect the corresponding module.

To keep apart the daemon's configuration options from the modules' configuration options the nodes of the configuration file have to differ. Configuration options for the daemon are children nodes without attributes. Configuration options for modules are encapsulated by children nodes with the name `module` and the additional attribute `token`. The following example is based on the already discussed demonstration module (§5.3). It shows one daemon option `daemon_option` and two module options (`disable` and `verbose`) for the module with the token `module_example`:

```
<ialdconfig>
    <!-- Daemon option -->
    <daemon_option>value</daemon_option>

    <!-- Module options -->
    <module token="example_module">
       <disable>false</disable>
       <verbose>true</verbose>
    </module>
</ialdconfig>
```

The parser compares the daemon's static configuration options with the expected configuration options. If a valid configuration option is found, the parser is responsible for taking the corresponding action—e.g. set the value of a variable to the obtained value. The dynamic configuration options for modules are handled by the function `conf_parse_module()`. This function is invoked by `conf_parse_file()` if a child node with the name `module` and the attribute `token` is found. The function `conf_parse_module()` iterates the double linked list which contains all available modules, and looks up the token `token` of each module. Once `token` matches with `token` read from the configuration file, `conf_parse_module()` iterates through the module's options and compares the options' names with all children nodes' name. If a match is found, `conf_parse_module()` allocates memory and stores the value for the corresponding option. It is each module's responsibility to check whether the values of the configuration options set by the daemon's configuration file parser are reasonable.

The nested, rather complex implementation of the function `conf_parse_module()` is illustrated using C pseudocode. The pseudocode shows the various iterations and the performed comparison operations:

```
module = module_list_head;

/* Iterate all modules, start at the list head. */
while(module) {
  if(node.token == module.token) {
    child = node.child;
    /* Iterate all module options for each child of the node. */
    while(child) {
      while(option) {
        if(child.name == option.name) {
          /* Match found. */
          option.value = child.value;
        }
        option = option->next;
      }
      child = child->next;
    }
  }
  module = module->next;
}
```

Each time the configuration file parser hits upon a configuration option which does not match with a daemon or a module option it issues a corresponding warning using the library's logging system. Stage two of the daemon's start-up is completed once the configuration file parser finished parsing the configuration file.

Command Line Interface

The command line interface is directly implemented by the daemon (`iald.c`) and serves two different purposes: On the one hand, the command line interface is responsible to implement a command line parser which evaluates the supplied command line options. On the other hand, it is responsible to provide a user space interface which gives information about the possible command line options. Again, the configuration options can be divided into two categories: configuration options for the daemon and for the modules. The parsing of the daemon's static command line options is discussed briefly. The parsing of the dynamic command line options for the modules is discussed thoroughly.

The Input Abstraction Layer's command line interface follows the GNU coding standards regarding the implementation of command line interfaces [Fsf04a]. These standards are based on the "Utility Conventions" of the POSIX Specification [Iee04]. The GNU coding standards for command line interfaces follow an approach to make command line interfaces more user friendly by providing long-options. According to the POSIX Specification, command line options consist of a single letter: `program -h`, with `program` being the executable binary and `h` being a command line option. The GNU coding standards expand command line options by adding long-options: `program --help`, again with `program` being the executable binary and `help` being the command line option. According to the GNU coding standards, long-options are always prefixed by two dashes (`--`). Both kind of command line options can have additional arguments, e.g. `program -d 3` (`program --debug 3`) with `3` being the argument for the command line option `-d` (`--debug`).

Parsing of the command line options is realized using `getopt_long(3)`. The usage of the function `getopt_long()` requires the expected command line options to be known at compile time. Hence, this function can only be used to parse the daemon's configuration options. Each command line option is represented by a structure `option` which is declared as follows:

```
struct option {
    const char *name;
    int has_arg;
    int *flag;
    int val;
};
```

The member `name` of the structure is a string representing the name of the long-option. The value of `has_arg` indicates whether the configuration option requires an argument or not. If `has_arg` equals *0*, the configuration option does not take an argument, if `has_arg` equals *1* the configuration option needs an argument. Optional arguments are specified by setting `has_arg` to *2*. The member `flag` states how the result of the configuration should be returned. Usually `flag` equals *NULL* which means that `getopt_long()` returns the option's member `val`. The member `val` is commonly set to a single character which corresponds to the name of the long-option. All configuration options of the Input Abstraction Layer are stored in the array `long_options` of `option` structures. To indicate its end, the last element of the array has to be set to {`0, 0, 0, 0`}.

```
static struct option long_options[] = {
    {"debug",          1, NULL, 'd'},
    {"foreground",     0, NULL, 'f'},
    {"help",           0, NULL, 'h'},
    {"list",           0, NULL, 'l'},
    {"list-verbose",   0, NULL, 'L'},
    {"module-options", 1, NULL, 'm'},
    {"version",        0, NULL, 'v'},
    {0, 0, 0, 0}
};
```

For example, running the Input Abstraction Layer daemon with the long-option `--help` corresponds to the short command line option `-h`. Both cases cause the daemon to print its usage. The command line option `debug` (short option `d`) requires an argument which defines the logging priority (§5.2). If the option `foreground` (`f`) is set, the daemon does not detach itself on start-up. This function is not yet implemented. To get a list of the available modules, the daemon has to be started using the command line option `list` (`l`). Correspondingly, if the daemon command line option `list-verbose` (`L`) is supplied, the daemon prints a verbose list of the available modules and their configuration options. The configuration options for the modules are passed by the argument of the configuration option `module-options` (`m`). When called with the command line option `version` (`v`), the daemon prints its version number.

For each configuration option the command line parser invokes a corresponding function.

These functions are responsible for evaluating the supplied configuration options. The Input Abstraction Layer's most important functions for this task are the ones handling the command line options of the modules. They are invoked by the command line options `list`, `list-verbose` and `module-options`. The command line options `list` and `list-verbose` provide information about the modules and their possible configuration options.

If the configuration option `list` (short option `l`) is supplied, the function `opt_list()` is invoked. This function iterates the double linked list and extracts the name of each module's data structure. The following source code shows a slightly shortened version of `opt_list()`:

```
/* Create a pointer to the list head. */
IalModule *module = modules_list_head;

/* Iterate all modules. */
while (module) {
    /* Print the name of the current module. */
    printf("%s\n", module->data->name);
    module = module->next;
}
```

The function `opt_list_verbose()` is called if the command line option `list-verbose` (short option `L`) is supplied. It is implemented similarly to `opt_list()`. Additionally, it prints the possible configuration options of the modules by iterating each module's options. To accomplish this approach, pointer arithmetic is used. The following example shows the iteration of the options array for one module:

```
/* Create a pointer to the module's option array. */
ModuleOption *option = module->data->options;

while(option->name) {
    /* Print the description, name and default value of the module option. */
    printf("%s, default: %s=%s\n", opt->descr, opt->name, opt->value);

    /* Go to the next element of the options array. */
    *option++;
}
```

Configuration options for Input Abstraction Layer modules are parsed by the function `opt_modules_opts()`. This function is invoked if the daemon is started with configuration option `module-options` (short option `m`). This configuration option requires an argument `mod_options`, which is a string. This string contains the configuration options for one or more modules. The format of the string `mod_options` is defined as:

```
mod_options := <mod_option>[,<mod_option>]
mod_option  := <token>:<option>=<value>[:<option>=<value>]
token  := Module token
option := Option name
value  := Option value
```

The string mod_options consists of one or more strings with the format mod_option. The function opt_modules_opts() parses the string mod_options using string operations provided by the GNU C Library. One mod_option supplies the configuration options for one module. To pass configuration options for several modules, a comma-separated list of mod_option strings has to be passed as argument. The following example is based on the already discussed demonstration module (§5.3). For this module, a valid argument for the configuration option module-options would be:

```
example_module:disable=false:verbose=true
```

Granted that a second module with the token example_module2 is available which has same options as the first demonstration module, a valid argument for module-options would be:

```
example_module:disable=false:verbose=true,example_module2:disable=true
```

Possible configuration options which were already set by the configuration file parser (stage two) are overwritten by the command line parser. This is done on purpose as it offers an easy way for testing various configuration options of the daemon without the need of changing the Input Abstraction Layer's configuration file. As soon as the command line parser has finished processing the command line options, the third stage is finalized. At this point the set-up of all modules is completed. The daemon is set up, too, but the D-BUS environment has still to be set up before it is possible to start the modules.

D-BUS

The daemon performs several operations to set up the D-BUS environment for the Input Abstraction Layer at stage four. First of all, the daemon's global variable dbus_connection of type DBusConnection represents the connection to the system message bus. To establish a connection to the system message bus, the daemon calls the function ial_dbus_connect(), which is implemented by the library (§5.2). The daemon subsequently creates the D-BUS service using this connection. The following source code shows the set-up of the D-BUS service:

```
/* Establish connection to the system message bus. */
if (ial_dbus_connect() == FALSE) {
    ERROR(("D-BUS connection failed."));
    exit(1);
}

/* Create D-BUS interface "com.novell.Ial". */
dbus_bus_acquire_service(dbus_connection,
                         "com.novell.Ial",
                         0,
                         &dbus_error);
```

The first argument of dbus_acquire_service() represents the connection to the D-BUS. The string com.novell.Ial passed as second argument states the name of the D-BUS service which is requested for creation. No additional flags are passed since the third argument is set to *0*. If the D-BUS daemon is not able to create the requested service, the structure `dbus_error` is filled with information. This structure is of type `DBusError` and represents a D-BUS exception.

Module Invocation

Once the connection to the system message bus is established and the service is acquired, the daemon is completely initialized. At this point, it is safe to start all available modules. This approach is realized by the function `modules_load()`, which is implemented by the module loader. The essential part of the function's source code follows:

```
/* Create a pointer to the list head. */
IalModule *module = modules_list_head;

/* Iterate all modules. */
while (module) {
    if (module->data->load() == FALSE) {
        /* Module failed to load. */
        WARNING(("Failed to load %s.", module->data->name));
    }
    else {
        /* Module successfully loaded. */
        INFO(("%s loaded.", module->data->name));
    }
    module = module->next;
}
```

The `while` loop iterates the module list and invokes the loading function of each module. The member `load` (`module->data->load`) of each module's `ModuleData` data structure contains a function pointer to its loading function. It is each module's responsibility to perform the following three steps at this point:

- Verify module options
- Initialize module variables and kernel interface
- Return *TRUE* on success and *FALSE* in case of failure

The module's loading function has to check carefully if the configuration file parser and the command line parser have initialized the module's options correctly. If they are set up correctly, the values of the options have to be assigned to the corresponding variables of the module. Subsequently, the module has to check whether it is able to access the kernel interface for which the module is designed for. If everything succeeds, the loading function has to return *TRUE*. Otherwise, *FALSE* has to be returned. As soon as all available modules are invoked, stage five is completed.

Event Loop

The start up of the daemon is completed by entering the main event loop. This event loop is implemented by the GLib library. In combination with functions provided by GLib it is possible to add callback functions to the event loop. A callback function is invoked as soon as a specified event occurs. The source of an event can be a timeout or the modification of a file descriptor (file, pipe or socket).

If a module needs to poll a kernel interface, it needs to add its callback function to the event loop by using a timeout mechanism. This is accomplished by using GLib's function `g_timeout_add()`:

```
g_timeout_add(guint interval, GSourceFunc function, gpointer data);
```

The argument `interval` states the interval between the event loop calls the function passed by the argument `function`. The GLib data type `guint` corresponds to the standard C `unsigned int` type. Each time the event loop calls the callback function, it passes the `gpointer data` to the function. The GLib data type `gpointer` corresponds to a standard C `void` pointer. This pointer is arbitrary—e.g. it can be a pointer to a data structure. Once the callback function is running it has to return either *TRUE* or *FALSE*. In the event the function returns *TRUE*, it will be invoked again as soon as the time specified by `interval` is elapsed. Once the callback function returns *FALSE*, it is removed from the event loop and will not be executed again.

Modules receiving input events from an interface which does not need to be polled use another approach to add callback functions to the event loop. GLib provides the data type `GIOChannel` representing an I/O channel. A `GIOChannel` is associated with a file descriptor referencing a file, pipe or socket. Various functions are provided by GLib in order to access data from a `GIOChannel`. A `GIOChannel` can easily be assigned as event source, too. This is achieved by defining a watch using the GLib function `g_io_add_watch()`. The combination of a `GIOChannel` and a watch offers a comfortable way to observe a file descriptor. Since various user space interfaces for input events are implemented either as file, pipe or socket this combination is very suitable for implementing an Input Abstraction Layer module. The following source code shows how a watch for a `GIOChannel` is added to the event loop:

```
/* Define file descriptor and GIOChannel. */
int event_fd;
GIOChannel *io_channel;

/* Create file descriptor for /dev/interface and associate it with io_channel. */
event_fd = open("/dev/interface", O_RDONLY);
io_channel = g_io_channel_unix_new(event_fd);

/* Add a watch of the io_channel to the event loop. */
g_io_add_watch(io_channel, G_IO_IN | G_IO_ERR, event_callback, data);
```

The file `/dev/interface` is a placeholder for an input event interface implemented as a file. After a file descriptor is acquired, it is assigned to `io_channel`. The watch is created by calling `g_io_add_watch()`. The supplied argument `io_channel` represents the `GIOChannel` which should be observed. The second argument states the conditions that have to be met to trigger an event. In the example two conditions are supplied: `G_IO_IN` and `G_IO_ERR`. The condition `G_IO_IN` is fulfilled if data is pending to be read from the `io_channel`. This implies that an event on the interface `/dev/interface` has happened. The second condition `G_IO_ERR` is fulfilled if an error occurred while reading from the `io_channel` and therefore, from `/dev/interface`. The argument `event_callback` is the callback function which is invoked by the event loop as soon as one of the given conditions is fulfilled. This callback function is called with the parameter data (`gpointer`).

Figure 5.7 gives an overview of the event loop, events and callback functions. One function (shown as "Function A") was added to the event loop using `g_timeout_add()`. The other function ("Function B") was added using the combination of a `GIOChannel` and a watch created by `g_io_add_watch()`. On each iteration, the event loop evaluates if the interval for "Function A" has elapsed. Once the time has elapsed, the function is invoked and the event loop continues its iteration afterwards. Otherwise, the event loop continues without invoking "Function A". The same procedure applies to "Function B": if a condition for the `GIOChannel` is fulfilled, "Function B" is invoked. Otherwise, the event loop continues on.

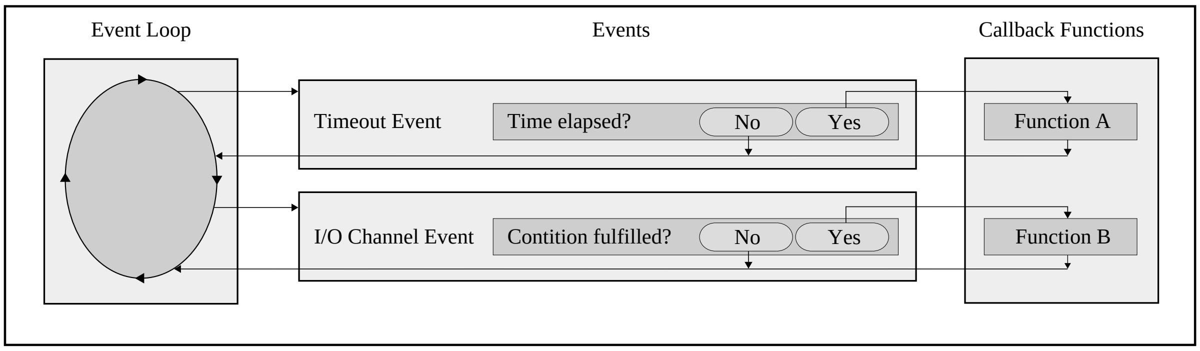


Figure 5.7: Input Abstraction Layer Event Loop

5.5 Event Interface Module

The event interface module `libial_evdev` is responsible for reporting any input events triggered by an input device driver using the Linux input core (Chapter 2, §2.1). On that ground, this module is responsible for generating abstract input events for the following input devices:

- Regular keyboards (Chapter 2, §2.2)
- USB keyboards (Chapter 2, §2.3)
- Bluetooth keyboards (Chapter 2, §2.4)
- Acer and Hewlett Packard function keys (Chapter 2, §2.7)

The reporting of keyboard input events realized with this module is most essential. As pointed out in Chapter 2, §2.2 the keyboard handler only reports input events for which a translation from scan- to keycode is available. But most Linux systems do not have a translation table for special and multimedia keys by default. The event interface closes this gap by reporting all input events of keyboards—whether there is a translation available for a key's scancode or not. The source code of the module is found in the directory `modules`. The prefix `libial_evdev_` is common for all files belonging to the module.

The event interface module can be enabled or disabled. This is the only configuration option provided by the module. The following source code shows a shortened version of the module's relevant data structures:

```
ModuleOption mod_options[] = {
    {"disable", "false", "disable=(true|false)"},
    {NULL}
};

ModuleData mod_data = {
    .name = "Event Interface Input Abstraction Layer Module",
    .token = "evdev",
    .options = mod_options,
    .load = mod_load,
    .unload = mod_unload
};
```

At initialization time the module first checks its configuration option. If the option `disable` is set to *true*, it returns *FALSE* to signal the daemon that the initialization was not successful. Otherwise, the module tries to initialize the event interfaces. For this purpose, the module scans the directory `/dev/input/` for available input event interfaces—each one representing one input device. Furthermore, each applicable input device is observed for input events. Once an input event occurs, the module translates the concrete input event to a corresponding abstract input event. The abstract input events are reported by utilizing the library's function `event_send()`.

The module's loading function is invoked by the daemon. The module's option `disable` is verified by a string comparison using `strcmp(3)`. If the module is not disabled, it proceeds. The verification of each available interface is divided into several steps. First, the module tries to get a file descriptor for all possible event interfaces. This is done by a loop which iterates all event interfaces from `/dev/input/event0` to `/dev/input/event31`. If the module has successfully acquired a file descriptor, it performs a check to determine whether the protocol of the event interface matches the protocol known to the module. This is done using the event interface's `ioctl(2)` request `EV_VERSION`. If this check is being accomplished without an error, the module further creates a `GIOChannel` and associates a corresponding watch with the daemon's event loop. The callback function `evdev_callback()` is common for all watches. Each watch is added with the two conditions `G_IO_IN` and `G_IO_ERR`.

At this point the module's initialization is complete. As soon as an input event occurs, the condition `G_IO_IN` is fulfilled and the callback function `evdev_callback()` is invoked by the event loop. The function is called with one argument: the file descriptor of the event interface on which the event occurred. Once called, the callback function has several responsibilities:

– Read data from the file descriptor
– Check whether it was a keypress
– Check whether the key is blacklisted

– Generate an abstract input event
– Send the abstract input event

Reading data from the event interface returns a structure `input_event` (Chapter 2, §2.1). The structure's member `type` gives information whether the occurred input event was a keypress or not. A keypress' `type` has the value `EV_KEY`. For security reasons, the module only reports input events of keys which are not blacklisted. Beside others, blacklisted keys are all alpha numeric keys. This prevents processes from abusing the Input Abstraction Layer to intercept passwords or other sensitive data. Additionally, the blacklist contains keys which are not found on keyboards. Such keys encompass buttons of input devices such as mice and joysticks.

This blacklist is implemented by the following macro:

```
#define key_blacklisted(code) \
        ((code < KEY_MAX) && (code > KEY_MIN) ? FALSE : TRUE)
```

The scancode of the pressed key is stored in `code`. `KEY_MIN` and `KEY_MAX` limit the range of scancodes which are not blacklisted. All other keys are blacklisted. The value of `KEY_MIN` is set to 0x*32* and the value of `KEY_MAX` is set to 0x*1ff*. Thus, the macro `key_blacklisted()` returns *FALSE* only in case the value of `code` being greater than `KEY_MIN` and less than `KEY_MAX`. Otherwise, `key_blacklisted()` returns *TRUE*. The lower boundary defined by `KEY_MIN` blacklists all scancodes of alpha numeric keys. The upper boundary `KEY_MAX` blacklists scancodes of keys and buttons which are found on other input devices than keyboards.

If an input event is not blacklisted, the module continues its processing: an abstract input event using the data structure `IalEvent` is created and afterwards sent to the Input Abstraction Layer's output interface. The member `sender` is set to the token of the module (`evdev`).

The data structure's member `sender` is set to the name of the input event interface which triggered the event. This name is acquired using `ioctl(2)`. The corresponding `ioctl` request is `EVIOCGNAME`. It is defined by the event interface driver and returns the name of a device. The following example shows how the name of a device (`/dev/input/event0`) is obtained:

```
int fd = open("/dev/input/event0", O_RDONLY);
char device_name[128];

if (ioctl(fd, EVIOCGNAME(sizeof(device_name)), device_name) == -1) {
    strcpy(device_name, "Unknown device");
}
```

If the `ioctl` request succeeded, `device_name` now contains the name of the device `event0`. Otherwise—if the request failed—the string *Unknown device* is copied to `device_name`. The event's member `name` is set to a descriptive string obtained by translating the scancode. This descriptive string is stored in a translation table of the module. To translate a scancode to its corresponding description, the module's macro function `key_to_string()` is used. The scancode has to be supplied as parameter. The return value of the function is a pointer to the descriptive string (`char *`). At last, the member `raw` is set to the scancode of the input event.

Once the data structure is filled, the module sends the abstract input event by calling the library's function `event_send()`. Correspondingly, the event is delivered to the D-BUS interface of the Input Abstraction Layer.

The creation and sending of an abstract output event for a concrete input event is illustrated by the following example. Granted that an input device has the name "AT Translated Set 2 keyboard" and offers a special key indicating a symbol for "E-Mail". The key's scancode equals 0x*9b*. Once this key is pressed, the driver for AT and PS/2 keyboards (Chapter 2, §2.2) generates an input event which is processed by the input core. The event interface driver gets notice of this event since it has registered itself as an event handler. Upon being invoked by the event loop, the callback function evaluates the input event. Since the input event is a keypress event and the event is not blacklisted, the callback function continues processing the event. An abstract input event `event` of the type `IalEvent` is created and the members of the structure are set-up:

- `event.sender` is set to the module's token (*evdev*)
- `event.source` is set to the input device's name (*AT Translated Set 2 keyboard*)
- `event.name` is set to the key's description (*E-Mail*)
- `event.raw` is set to the scancode (0x*9b*)

5.6 ACPI Module

The Input Abstraction Layer's ACPI module `libial_acpi` reports abstract input events for ACPI events. Input events reported by the ACPI subsystem have been discussed in Chapter 2, §2.6. The fact that the ACPI module reports input events on all ACPI compliant computers—desktop and mobile systems—underlines its relevance. The module's source code is found in the directory `modules`. The prefix `libial_acpi_` is common for all files of the ACPI module.

The ACPI subsystem reports more than just input events. The ACPI module has to sort out the relevant input events. Such as the events of the power button, sleep button and lid switch. The following source code shows a shortened version of the modules' relevant data structures:

```
ModuleOption mod_options[] = {
    {"disable", "false", "disable=(true|false)"},
    {NULL}
};

ModuleData mod_data = {
    .name = "ACPI Input Abstraction Layer Module",
    .token = "acpi",
    .options = mod_options,
    .load = mod_load,
    .unload = mod_unload
};
```

The module offers the configuration option `disable` which specifies whether the module is enabled or disabled. The module's loading function evaluates the value of `disable`. By default the module is enabled. Further, the loading function initializes the user space interface of the ACPI subsystem if `disable` equals *false*. This initialization routine first tries to access the user space interface `/proc/acpi/event` directly. If this approach fails, it is most likely that another process has already acquired the interface. If this process is the ACPI daemon `acpid`, another user space interface is available to receive the ACPI events. The ACPI daemon creates the socket `/var/run/acpid.socket` to which it directly passes all ACPI events.

If the module is able to access either of the two interfaces, it subsequently creates a `GIOChannel` and adds a watch to the event loop. The watch is added with the conditions `G_IO_IN` and `G_IO_ERR`. The callback function of the watch is `acpi_event_callback()`. Once an ACPI event occurs, the callback is invoked and gathers information about the ACPI event using the `GIOChannel`. It does not matter whether the `GIOChannel` is associated with a file descriptor of `/proc/acpi/event` or with the socket of the ACPI daemon. The `GIOChannel` provides transparent access to the associated data source by using functions of the GLib library. The ACPI module uses the GLib function `g_io_channel_read_line_string()` to read data from the module's `GIOChannel`. The function `g_io_channel_read_line_string()` stores the read data as a string. This string contains the complete ACPI event.

The module has to parse the string of the ACPI event to verify whether the event actually is an ACPI input event from one of the buttons or the lid switch. First, the string is examined to determine if it contains the substring `button`. As previously discussed, all ACPI input events are prepended with this substring (Chapter 2, §2.6). If the string does contain `button`, the parsing continues. Otherwise, the ACPI event is not a button event. To determine which button was actually pressed, the string is further examined to see if it contains one of the three substrings: `PWRF` (power button), `SLPF` (sleep button) or `LID` (lid switch). In case of the power button and the sleep button, the module calls the module's function `acpi_event_send()` which performs the abstraction of the concrete event.

Otherwise—in case of the lid switch—another function of the module (`acpi_lid_state()`) is invoked before sending the input event. Since the ACPI input events for opening and closing the lid are equal, it is not possible to determine whether the lid was opened or closed. However, this information can be obtained through the ACPI button driver's user space interface. The function `acpi_lid_state()` evaluates the lid state by reading the `proc` entry `/proc/acpi/button/lid/LID/state`. If the read value is *0* the lid is closed. Correspondingly, if the value equals *1*, the lid is open. Once the function has gathered this information, it invokes the creation of the corresponding abstract input event by calling `acpi_event_send()`.

The function `acpi_event_send()` requires a string argument (`char *button`) which represents the button evaluated by the parser. The argument `button` is set to *Power Button* for the power button, and it is correspondingly set to *Sleep Button* for the sleep button. A lid switch event is reported by setting `button` to *Lid Switch (open)* if the lid was opened and to *Lid Switch (close)* if the lid was closed.

The creation of the abstract input event is performed by `acpi_event_send()` using the structure `IalEvent`. The members of the structure are set up as follows:

- `event.sender` is set to the module's token (*acpi*)
- `event.source` is set to */proc/acpi/event*
- `event.name` is set to the value of `button`
- `event.raw` is set to *-1*

The member `raw` is not available for ACPI events since they have no numeric representation. Hence, the value of `raw` is set to *-1* for all events. Once the abstract input event `event` is set up by `acpi_event_send()`, it sends the event to the Input Abstraction Layer's output interface by invoking the library's function `event_send()`.

5.7 Toshiba Module

The Toshiba module `libial_toshiba` is applicable for most recent Toshiba laptops (Appendix C, §C.8). It translates input events of function and multimedia keys found on these laptops to abstract input events. It depends on the Toshiba ACPI driver discussed in Chapter 2, §2.6. The Toshiba module needs to poll the user space interface of the driver to receive input events. The prefix `libial_toshiba_` is common for all files belonging to the Toshiba module. It resides in the directory `modules`. The following source code is an excerpt of the Toshiba module's data structures:

```
ModuleOption mod_options[] = {
    {"disable", "false", "disable=(true|false)"},
    {"poll_freq", "250", "poll_freq=n (n: polling frequency in ms)"},
    {NULL}
};

ModuleData mod_data = {
    .name = "Toshiba Input Abstraction Layer Module",
    .token = "toshiba",
    .options = mod_options,
    .load = mod_load,
    .unload = mod_unload
};
```

The module has two configuration options: `disable` and `poll_freq`. The option `disable` defines whether the module is enabled or disabled. By default, `disable` is set to *false*. The second configuration option, `poll_freq`, specifies the interval the module is polling the driver's user space interface. The measuring unit for `poll_freq` is the millisecond. By default, the module polls the driver's interface every 250 milliseconds. This value exceeds the performance requirement (Chapter 3, §3.3). Choosing a shorter polling interval improves the latency between the user action and the delivery of the input event. In the course of choosing a shorter polling interval, the latency is lowered on the expense of system resources.

Since the module relies on polling the interface it can not define a combination of a `GIOContainer` and a watch, which is associated with the event loop. The module is required to register a callback function utilizing the function `g_timeout_add()` provided by GLib. The callback function `toshiba_key_poll()` is added to the event loop with the interval specified by the configuration option `poll_freq`.

Each time `toshiba_key_poll()` is invoked, it evaluates if a function key event has happened. This evaluation is realized by the module's function `toshiba_key_ready()`, and implements the procedure described in Chapter 2, §2.6. If a function key event was obtained, the module verifies if a translation for the function key event is available. This translation is similar to the translation of a scancode value to its corresponding description. The values for the function events received by polling the Toshiba ACPI driver's user space interface are reported as a unique numeric value.

The structure `Key` represents a function key combination. The structure has two members: `value` and `descr`:

```
struct Key {
    const int value;
    const char *descr;
};
```

The value corresponds to the numeric value of the function key combination and `descr` is a string containing a description for the function key. The second, static data structure `keys` required for the translation of a function key event is an array with elements of type `Key`:

```
static struct Key keys[] = {
    {0x100, "Fn-Escape (Mute)"},
    {0x101, "Fn-1 (Volume Down)"},
    {0x102, "Fn-2 (Volume Up)"},
    ...
    {0xb85, "Hotbutton (TV-Button)"},
    {0xb86, "Hotbutton (E-Button)"},
    {0xb87, "Hotbutton (I-Button)"},
    {0, NULL}
};
```

This array is iterated by the module's function `toshiba_fnkey_description()` to gather a function key's description. Once a function key event has occurred and the corresponding description is acquired, `toshiba_key_poll()` calls `toshiba_event_send()`. This function creates an abstract input `event` (type `IalEvent`) and sends the event to the Input Abstraction Layer's output interface. The members of the abstract input event `event` are set up with the following values:

– `event.sender` is set to the module's token (*toshiba*)

– `event.source` is set to */proc/acpi/toshiba/keys*

– `event.name` is set to the description of the function key

– `event.raw` is set to numeric value of the function key

For example, if the function key `Fn-Escape` is pressed, the module obtains the value 0x*001*. The description *Fn-Escape (Mute)* for this function key is acquired afterwards. At last the abstract input event is created and filled with the values common for all function keys (`sender` and `source`) as well as with the unique values for `name` (0x*001*) and `raw` (*Fn-Escape (Mute)*).

Chapter 6

Requirements Verification

This chapter compares the achieved goals with the requirements specification defined in Chapter 3. First, the verification of the functional requirements is performed. This part of the verification compares the implementation with the demanded functional requirements. Secondly, the performance requirements are verified. In order to verify whether the performance requirements are met as demanded, a kernel driver was implemented to measure the latency of input events. At last, the quality and security requirements are verified.

6.1 Functional Achievements

The first functional requirement demands a utilized user space interface to report up to now unattended input events. This requirement is fulfilled by the implementation of the Input Abstraction Layer's abstract output interface. This interface is shared by all modules to report input events of the kernel drivers generating various input events.

The implemented output interface of the Input Abstraction Layer does not require any privileges to be utilized by any user space application. This fact accomplishes the second functional requirement which specifies that the existing barriers have to be eliminated, with one restriction: the Input Abstraction Layer itself has to run with `root` privileges, as some kernel interfaces have access restrictions as defined in Chapter 3, §3.1.

The unified data representation demanded by the third functional requirement is achieved by the Input Abstraction Layer's data structure `IalEvent`. It represents all kinds of input events in a common way. The conversion of the concrete input events to the unified representation is performed by the Input Abstraction Layer's modules.

The fourth functional requirement asks for a permanent location of the abstract output interface. The location of the implemented output interface depends on D-BUS to a large extent. If the application programming interface of D-BUS changes, it is unavoidable that the Input Abstraction Layer's output interface changes, too. Thus, this functional requirement is only fulfilled as long as the Input Abstraction Layer does not change the name of the interface, and the D-BUS' application programming interface does not change.

The last functional requirement specifies that the abstract output interface has to be independent in regards to the programming language of the application which wants to utilize the interface. The requirement is entirely fulfilled since D-BUS offers bindings for many recent programming languages and frameworks, such as C, C++, Mono (e.g. C#), Qt (C++), Perl, Python, Ruby and Java.

6.2 Performance

The performance requirements specify that the Input Abstraction Layer should only use a reasonable amount of system resources while running. To evaluate the consumption of system resources, the Input Abstraction Layer's run-time behavior was observed with `top(1)`. This program provides a dynamic real-time view of the processes' resource consumption running on a system. The common requirements such as low memory usage and low CPU utilization are fulfilled. The daemon does not have a negative impact on the general system performance. The event loop of the Input Abstraction Layer's daemon ensures the reliable delivery of input events occurring on the different input interfaces. The user space applications using the Input Abstraction Layer's output interface to gather input events do not have to poll the interface which is demanded by the performance requirements, too. Applications can utilize several D-BUS bindings to create callback functions once an input event is sent to the D-BUS interface of the Input Abstraction Layer.

To evaluate the period of time an input event takes to reach user space applications, the kernel driver `iallatency` (Appendix D, §D.3) has been implemented. Using this driver, it is possible to measure the time from the input device driver to the receiving of the event by a user space application. Several internals of the Linux kernel have to be discussed first in order to understand the approach of the implemented driver.

Depending on the input device's implementation, input events are reported either by generating an interrupt or by storing the event in a specific hardware register, which needs to be polled. Interrupt-driven devices are generating interrupts once an input event happens. This interrupt is relayed by the device's controller to the system's interrupt controller which sends a signal to the processor. Subsequently, the processor notifies the Linux kernel about the interrupt, which handles it appropriately. The interrupt handling in Linux 2.6 is described in depth in Chapter 5 of [Lov03]. For example, if a keypress on a regular keyboard is issued, the keyboard controller generates an interrupt. The processor receives the interrupt and notifies the Linux kernel about the interrupt. The kernel correspondingly calls the interrupt handler for the occurred interrupt.

Devices which do not generate interrupts need to be polled. This polling can either be initiated in kernel or user space. Device drivers which are polling in kernel space use kernel timers for this approach. Beside others, a kernel timer (declared in `<include/timer.h>`) has the members `expires` and `data`. The member `expires` specifies the point of time at which the callback function defined by `data` is executed. Kernel timers are controlled by kernel's interrupt handler for the timer interrupt. If a device driver does not use kernel timers to poll the device, the driver is required to export a user space interface to implement the polling. For example, the driver creates an interface using the `proc` file system. Once a user space process accesses this interface by issuing a system call (`open(2)`, `read(2)`), a corresponding function of the kernel driver is executed.

The addressed timer interrupt occurs every 1/`HZ` second. The kernel defines the `HZ` default value to *1000* on the i386 architecture—the timer interrupt runs each millisecond. During each timer interrupt the global kernel variable `jiffies` of type `unsigned int` is incremented by one. Since `jiffies` is initialized with *0* at boot time, the system's uptime can be calculated with the values of `jiffies` and `HZ`:

$$uptime = \frac{\text{jiffies}}{\text{HZ}}\text{s}$$

Figure 6.1 gives an overview of the time needed by an input event to reach a user space application. The user action on an input device triggers a hardware event which attains kernel space—the input device's driver—either by generating an interrupt or by being polled. This period of time is denoted as t_1. The time interval between the reception of the input event by the input device's driver, and the actual receiving by an application in user space, is denoted as t_2. Thus, the time t_{total} an input event needs to reach a user space application equals to:

$$t_{total} = t_1 + t_2$$

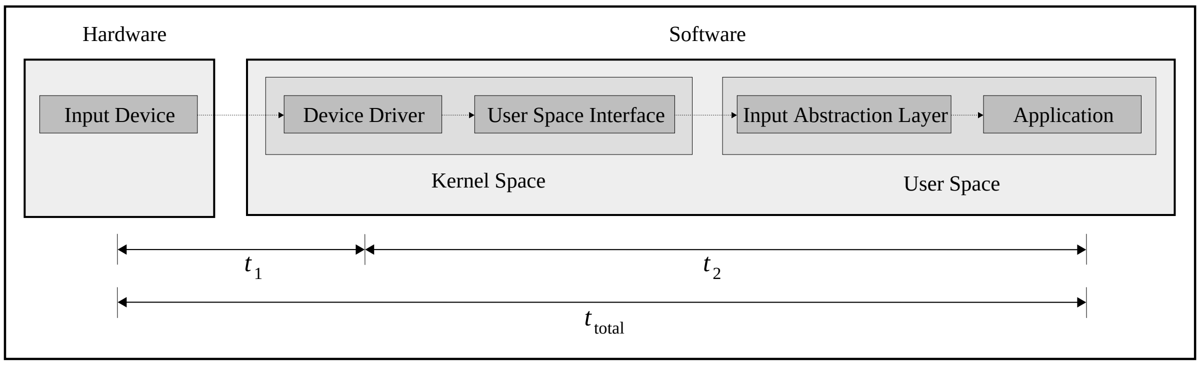


Figure 6.1: Time Interval for Input Events to Reach User Space

The time interval t_1 heavily depends on the kind of the input device. An input event of an interrupt-driven input device has a t_1 less than an input event of a polled input device. In general, t_1 for interrupt-driven devices can be neglected. It is of the order of magnitude of one millisecond. The time interval t_1 of a polled input device depends on how often the polling is executed. Hence, for polling devices, no general predication for the actual value for t_1 can be made.

The second time interval t_2 is determined the kernel driver `iallatency`. The first measuring point is set in kernel space at the time the driver of an input device gets notice of the occurrence of an input event. The second measuring point is invoked by a user space application right after receiving the input event. At both measuring points, the current value of `jiffies` is stored. Thus, the measurement of t_2 is accurate with a one millisecond tolerance. Beside the implementation of the kernel driver, several parts of the kernel itself had to be modified to accomplish the measurement:

– Definition of a global variable used to store the first measuring point

– Modification of the input device drivers to store the first measuring point

The global variable was added to `init/main.c` of the kernel source code. The following source code shows the additional code:

```
unsigned long event_jiffies = 0;
EXPORT_SYMBOL(event_jiffies);
```

The variable `event_jiffies` of type `unsigned long` is initialized with a zero. To make the variable accessible to other kernel drivers—the input device drivers—its symbol needs to be exported. This is done with the macro `EXPORT_SYMBOL`.

To measure t_2, the input device drivers are required to store the current value of `jiffies` into `event_jiffies` once an input event occurs. To do so, the driver needs to include `include/jiffies.h` and declare `event_jiffies` as an external variable:

```
#include <linux/jiffies.h>
extern unsigned long event_jiffies;
```

While this code is common for all input device drivers for which t_2 should be determined, the actual storage of the `jiffies`' value highly depends on the device driver. Three different drivers have been modified in order to measure t_2:

- AT and PS/2 keyboard driver (Chapter 2, §2.2)
- ACPI button driver (Chapter 2, §2.6)
- Toshiba ACPI driver (Chapter 2, §2.6)

The AT and PS/2 keyboard driver `atkbd.c` was modified in the following way: as soon as the keyboard interrupt controller i8042 issues an interrupt, the interrupt handling routine `atkbd_interrupt()` of `atkbd.c` is invoked by the Linux kernel. This routine generates the input event for the Linux input core by processing the data received from the keyboard. Right before the actual input event is reported, the driver stores the actual value of `jiffies` to `event_jiffies`. To avoid the `jiffies` to be stored on every keypress, it previously checks if the scancode of the pressed key was 0x*57*. This value corresponds to the key F11.

Regarding the ACPI button driver `button.c`, a similar modification was implemented. Before the button driver invokes the ACPI bus drivers' function `acpi_bus_generate_event()` to report that an ACPI button was pressed, the `jiffies` are stored to `event_jiffies`. This time, the value of `event_jiffies` is stored whenever an ACPI button is pressed. No specific button is sorted out before saving the current `jiffies`.

The Toshiba ACPI driver's function `read_keys()` is executed once a process is reading from the driver's user space interface `/proc/acpi/toshiba/keys`. This routine gathers data from the system event register. If a function key was stored in the register, the `jiffies` are stored into `event_jiffies` correspondingly. This time, the `jiffies` are only stored if the function key event was Fn-F1.

At this point, the device drivers are set up to store the first measuring point. The second measuring point is set by the implemented kernel driver. The driver `iallatency` (`init/iallatency.c`) exports two user space interfaces: `/proc/ial_event_received` and `/proc/ial_get_data`. Once a process issues the system call `open(2)` on one of the two interfaces, a corresponding function of the driver is invoked. In the event that `ial_event_received` is being accessed by a process, the current `jiffies` are stored to the variable `receive_jiffies` (type `unsigned_long`). Thus, this is how the second measuring point is set: a user space application is waiting for input events of the Input Abstraction Layer by utilizing its output interface. As soon as an input event occurs, the application invokes the storage of the second measuring point by issuing the following system call:

```
open("/proc/ial_event_received", O_RDONLY);
```

This system call causes `iallatency` to store the current `jiffies` into `receive_jiffies`. Once the the second measuring point is stored, it is possible to gather the two values of `event_jiffies` and `receive_jiffies` by accessing the second `proc` interface of the kernel driver:

```
open("/proc/ial_get_data", O_RDONLY);
```

This causes the kernel driver to report the values of `event_jiffies` and `receive_jiffies` by generating a kernel log entry. Once the measuring data are reported, the kernel driver sets both variables back to zero.

Both the modification of the kernel input drivers and the implementation of the kernel driver have been realized by causing as less overhead as possible. This ensures the measurement to be accurate. For example, it would have been possible to let the input device driver invoke `printk()` to issue a log entry to the kernel log in order to gather the first measuring point. Correspondingly, the user space application could report the current uptime of the system (second measuring point) once the input event is received. However, this approach would have implied too much overhead and thus, caused imprecise measurements.

All performed measurements have been accomplished on a system clocked at 750 MHz running the Linux kernel version 2.6.9. The tests were performed by gathering t_2 for 50 times at low system load and another 50 times at high system load. The average low system load was *0.65*. The high system load had an average of *2.42*. The value of the system load describes the average number of processes which are ready to run. The load *1.0* indicates that the system's CPU is fully utilized by the running processes. A load less than *1.0* means that the CPU is not fully utilized. Correspondingly, a load greater than *1.0* implies that the CPU is overwhelmed with work—more processes are ready to run than the CPU is able to serve.

The event interface has been investigated regarding the time an input event needs to reach user space. Figure 6.2 shows the results of the passes. At low system load, the following time intervals have been measured for t_2: t_{min} = 1ms and t_{max} = 8ms. The average time interval of all passes is t_{avg} = 1.94ms. The corresponding values for t_2 at high system load are: t_{min} = 1ms and t_{max} = 14ms, resulting in an average time interval t_{avg} = 4ms.

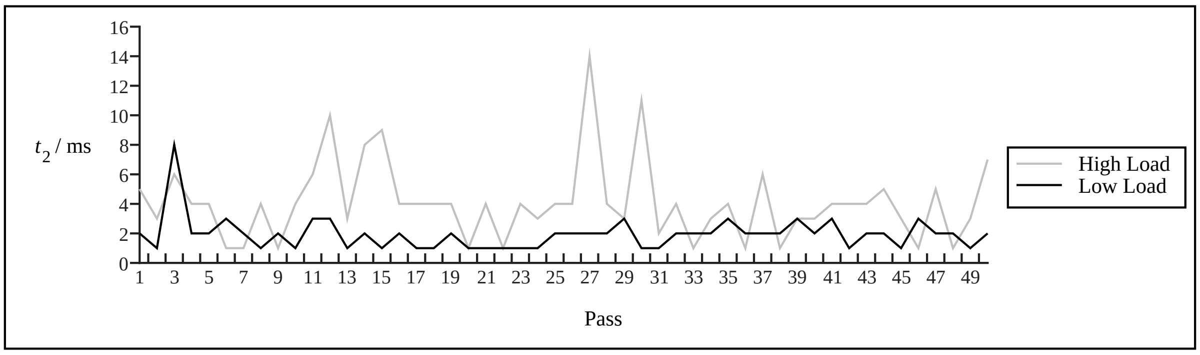


Figure 6.2: Keyboard Event Time Interval

The second test series was performed measuring the time intervals for ACPI buttons. Figure 6.3 shows the results of the passes for the ACPI button driver. At low system load, the minimum of t_2 was $t_{min} = 1$ms and the maximum was $t_{max} = 8$ms. The average time interval is $t_{avg} = 2.33$ms. The corresponding values for t_2 at high system load are: $t_{min} = 2$ms and $t_{max} = 17$ms. The average time interval is $t_{avg} = 6.51$ms.

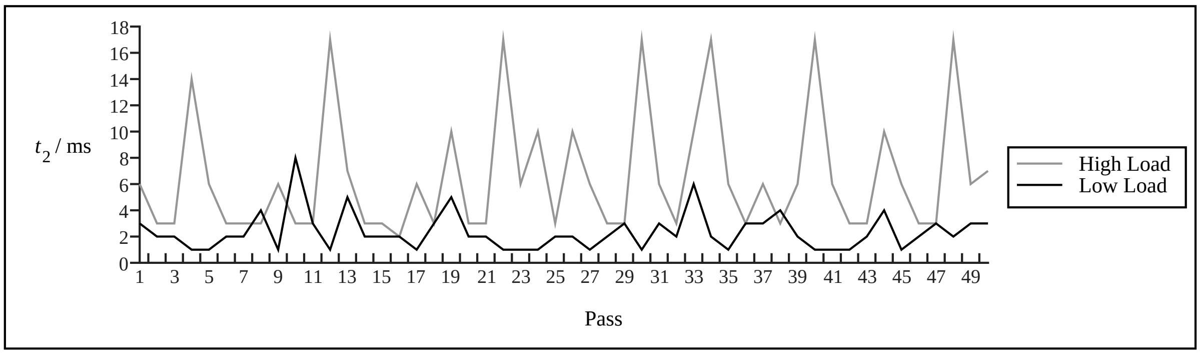


Figure 6.3: ACPI Button Event Time Interval

At last, the time intervals for events reported by the Toshiba ACPI driver have been determined. When running at low system load, the minimum of t_2 was $t_{min} = 0$ms and the maximum was $t_{max} = 5$ms. The result of the 12^{th} pass—which is the minimum—is a measurement error since it equals 0ms. This would indicate that there has been no delay between the kernel driver and the user space application. At high system load the results for the Toshiba ACPI driver are: $t_{min} = 1$ms and $t_{max} = 16$ms. The average time interval is $t_{avg} = 7.57$ms. Figure 6.4 shows the results.

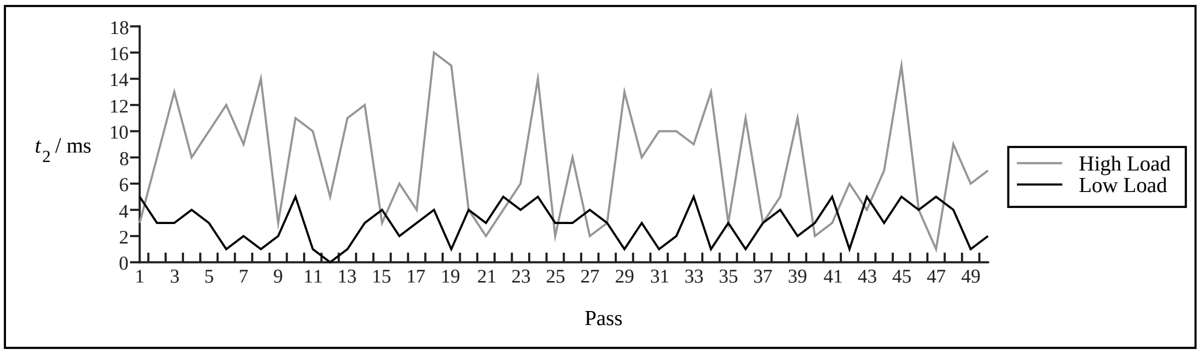


Figure 6.4: Toshiba ACPI Event Time Interval

The performance requirements (Chapter 3, §3.3) specify that the time between a user action and the arrival of the input event at the user space application should not exceed the limit of 25 milliseconds. To verify this requirement, t_{total} has to be calculated. For the keyboard driver and the ACPI button driver—they are both interrupt-driven—a worst case assumption $t_1 = 2$ms is made. The Toshiba ACPI driver uses polling. Hence, t_1 depends on the polling frequency and no general assumption for t_1 can be made. Together with the values for t_2 determined by the measurement series, t_{total} can be calculated. To determine whether the limit of $t_{total} = 25$ms is not exceeded, the measurement series' maximum of t_2 under high system load is used:

$$t_{total} = t_1 + t_{max}$$

The result for the keyboard events using the AT and PS/2 keyboard driver is:

$$t_{total} = t_1 + t_{max} = 2\text{ms} + 14\text{ms} = 16\text{ms}$$

The result for ACPI events processed by the ACPI button driver is:

$$t_{total} = t_1 + t_{max} = 2\text{ms} + 17\text{ms} = 19\text{ms}$$

The result for ACPI Toshiba drivers' input events depend on the polling frequency. Based on the assumption that the driver is polling every 250ms the corresponding result for t_{total} is at least:

$$t_{total} = t_1 + t_{max} = 250\text{ms} + 16\text{ms} = 266\text{ms}$$

Regarding interrupt-driven input devices, the results show that even in worst-case situations the limit of 25ms is not exceeded. Thus, the performance requirement is guaranteed for interrupt-driven input devices. In regard to input devices, which are polled by their device drivers, the limit of 25ms can not be redeemed since the polling intervals vary depending on the device driver. The performance requirement is not fulfilled. However, the Input Abstraction Layer does not delay the input event by a significant amount of time.

Since the Linux kernel does not meet hard real-time requirements, it is still possible that the time interval of input events of interrupt-driven devices exceed the limit of 25ms. Though, the measurements reflect that it is most likely that the limit is not exceeded—even under high system loads.

6.3 Quality and Security

The quality requirements described in Chapter 3, §3.4 are fulfilled: The logging system of the Input Abstraction Layer implements the demanded feature for error messages. The configuration is user-friendly and makes it possible to adjust the Input Abstraction Layer to each user's needs. The specified requirement to make the Input Abstraction Layer easily expandable is fulfilled by its modules. They can be added and removed without the need of recompiling the daemon.

The security requirement states that alpha numeric input events should not be processed to avoid any abuse of the Input Abstraction Layer. This responsibility is shifted from the daemon to the modules: they are responsible to prevent the reporting of alpha numeric input events. The blacklist functionality of the input event interface module fulfills this requirement.

Chapter 7

Open Source Development Methodology

The Input Abstraction Layer is entirely built upon Free and Open Source Software (FOSS). On the one hand, it depends on FOSS projects such as the C compiler of the GNU Compiler Collection (GCC), GLib and D-BUS. On the other hand, the Input Abstraction Layer itself is a FOSS project. This chapter outlines the development methodology of open source software on the basis of the Input Abstraction Layer. Important considerations of open source are explained in-depth.

Throughout this chapter the term FOSS is used in favor of *free software* or *open source software*. The difference between the two terms are of an ideological kind. The term free software was embossed by Richard Stallmann and the GNU project, while the term open source software was coined by Eric Raymond. Advocates of open source software can be seen as less radical as the protectionists of free software: the open source software community regards the issue whether software should be open source or not from a practical point of view. The free software community regards the issue as an ethical question. Beside their disagreements both communities have a common enemy: proprietary software.

FOSS has established a whole new basic approach on how software is developed. This approach focuses on a distributed network of developers and users who communicate with each other using the Internet to share thoughts, ideas and source code. Anyone interested can take part in this movement. This differs from the commercial approach where the development of software is done without comprising the users at all stages of the development. An in-depth look at the new approach on software development is given in [Ray01].

Taking the users into account for the development is one of the biggest advantages of FOSS. The users have a direct influence on the progress of the development by using the software and giving feedback to the developers. This ensures that software malfunctions are found quickly, and new features can be demanded by the users. Thus, the result of the developed software corresponds to what the users need.

Compared to closed source, companies using FOSS projects enjoy their independence since they do not depend on another company's business practices—especially another company's price policy. Even if the development of a FOSS project bogs down, the source code is available and another party can step into the project to support its development.

7.1 Licenses

All FOSS projects are released under the terms of a FOSS license. Both the free software movement and open source software movement have a list of approved licenses which they consider to match with their attitudes. The Free Software Foundation (FSF) provides a list of the qualified free software licenses in [Fsf04b]. A list of software licenses approved by the Open Source Initiative (OSI) is found in [OSS04].

Choosing an appropriate software license for a project is very important. All licensees of a software have to abide to the terms and conditions determined by the software's license. Companies are revising licenses precisely before deciding whether a certain software is used or not. This is required since a software's license determines how a licensee has to handle modifications and contributions to the software as well as the actual usage of the software.

For example, software which is released under a license which is compatible with the GNU General Public License (GPL) require the licensees to release modifications of the software, if any. Other licenses, such as the BSD license, enable the licensee to use and modify the software without releasing the modifications to the public.

To understand the meaning of the different kinds of licenses [Lau04] is a worthwhile book which discusses several free and open source licenses in a human-readable form. The book also describes different models of open source and free software development and how to choose the right license.

The Input Abstraction Layer is released under the terms of two licenses. The decision of releasing the software under two different licenses enables the licensee to choose under which conditions he is actually using the software. All parts of the Input Abstraction Layer are released under either of the following two licenses:

– GNU General Public License Version 2
– The Academic Free License Version 2.1

Dual licensing of the source code provides open and free access to the Input Abstraction Layer for both the free software community and developers or companies that cannot use the GPL [Fsf91]. The actual choice of the license is up to the licensee. If a licensee is able to publish modifications made to the Input Abstraction Layer, using the software under the terms of the GPL is reasonable. The GPL is accepted as a free software license by the Free Software Foundation, as well as it is accepted as an open source license by the Open Source Initiative.

Beside using the Input Abstraction Layer under the terms of the GPL, the licensee is free to use the Input Abstraction Layer under the terms of the Academic Free License [Ros04]. This license allows the usage and modification of the Input Abstraction Layer without the need to publish the changes. The Academic Free License includes a section (Paragraph 10) which determines that the license terminates once a licensee is suing the licensor for patent infringement:

> 10) Termination for Patent Action. This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any licensee alleging that the Original Work infringes a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware [Ros04].

This is the actual citation of Paragraph 10 of the Academic Free License. This clause should prevent a licensee from suing the licensor and using his work at the same time. The Academic Free License is accepted as an open source license according to the Open Source Initiative. The license is not classified as free software license by the Free Software Foundation since it is not GPL-compatible.

7.2 Tools and Services

The development of the Input Abstraction Layer has been accomplished by using free and open source software throughout. This section summarizes the most important tools and explains their meaning for the general development of FOSS. Additionally, several pointers for further reading are provided.

Prior to implementing the Input Abstraction Layer, a comprehensive examination of several open source projects has been performed. This way, sophisticated algorithms and thoughtful data structures are found. Even if a project has a completely new approach to solve a problem, the internals of the implementation are similar. Reading and understanding source code is one of the essential parts for a successful FOSS development. A helpful insight into reading foreign source code is provided by [Spi03]. This book gives an excellent insight on how to read code in general and examines several open source projects. The GNU Coding Standards [Fsf04a] are essential for the development of FOSS projects. These standards provide guidelines on software design, the use of the C programming language and on how software should be documented.

The GNU Autotools consist of the utilities `autoconf(1)`, `automake(1)` and `libtool(1)` which help to develop software that is platform and system independent to the greatest possible extent. Another reason to use the GNU Autotools are the packages for the various Linux distributions. Using the GNU Autotools spares the package maintainers a lot of time since dependencies to other programs or libraries are already defined by the `configure` script which is generated by `autoconf`. The use of the Autotools thus leads to a symbiotic relationship between the developers and the package maintainers: the developers relieve the package maintainers by supplying a standardized source package, the package maintainers take on the work to create the distribution's packages once a new version of the software is released.

The Autotools are available at `http://www.gnu.org/software/`. A comprehensive guide for working with the GNU Autotools is provided by [VET00]. The source code of the Input Abstraction Layer is released as a GNU Autotool package.

The tenor "release early, release often" [Ray01] is important to achieve a high quality level of the software. Publishing all changes of the software quickly after the implementation, malfunctions are found faster by the users and thus, they can be corrected before causing more trouble. It is suggestive to work on a version control system such as CVS or Subversion and allow public access to it. This way other developers and experienced users are given the chance to access the latest version of the source code. Stable branches of the development should be released as source packages.

CVS is available at `http://www.cvshome.org/`, Subversion can be obtained at `http://subversion.tigris.org/`. Further reading regarding the use of version control systems is provided by [FB03] and [CSF04]. Since subversion is relatively new, [CSF04] provides information about switching from CVS to Subversion. The development of the Input Abstraction Layer is realized using Subversion as version control system. Appendix D provides an example on how the Input Abstraction Layer's source code can be obtained using Subversion.

A software project always needs a decent documentation—for both end users and developers. The Input Abstraction Layer's source code is documented using Doxygen (Appendix D, §D.2). For this purpose, Doxygen is a powerful tool to generate the documentation while coding. The documentation is done by commenting the source code with a specific syntax. Doxygen afterwards parses the source code and generates the documentation. Beside others, it generates Unix man pages, browsable HTML pages and manuals in the PDF format. Doxygen is also able to generate graphs of dependency trees for the different source code modules and data structures. Doxygen and its documentation are available on the project's home page `http://www.doxygen.org/`.

7.3 Collaborative Environment

Several precautions have to taken to achieve a successful FOSS project. All of them have a common goal: communication. Communication between the project lead and other developers, as well as communication between the developers and users. The primary communication medium is email, either directly or by using mailing lists.

Users tend to have an inhibition threshold to contact developers directly using email. Therefore, it is good to have a web-based forum where users can post comments, questions, and can exchange their experiences with others. It is one of the most important tasks of the developers to monitor the forums' activity and help users who are having problems.

Capable users are a valuable resource for feedback and suggestions for the future development. It is often that users are developing new features on their own and supply so called patches which add new functionality to the project or fix malfunctions of the software. To avoid unintentional side effects, patches have to be reviewed carefully by the lead developers before they are applied to the source code.

To realize a central place to go for both developers and users, several services have to be set up. Services such as a project home page, a forum, mailing lists, a version control system and a bug tracking system. For this approach, the FOSS community can host their projects on centralized locations to control and manage the development:

– Berlios, `http://developer.berlios.de`
– Novell Forge, `http://forge.novell.com`
– Sourceforge, `http://www.sourceforge.com`
– Savannah, `http://savannah.gnu.org`
– Tigris, `http://www.tigris.org`

All of these web sites offer various services for FOSS projects for free. Thus, developers do not have to pay for network traffic, server hardware and system administration caused by the project.

Announcing a project's foundation and subsequent releases of the project is important to gain name recognition. For this purpose, it is reasonable to create project entries on web sites which are indexing FOSS projects. These web sites are visited by users looking for software. The following web sites offer developers to add projects to their index:

– Freshmeat, `http://www.freshmeat.net`
– Gnome Files, `http://www.gnomefiles.org`
– KDE Apps, `http://www.kde-apps.org`

The Input Abstraction Layer is hosted by Berlios. At present the following services have been set up. Additional services will be extended as required by the course of time:

– Home page, `http://ial.berlios.de`
– Doxygen documentation, `http://ial.berlios.de/doc`
– Subversion repository, `svn://svn.berlios.de/ial`
– Subversion web frontend, `http://svn.berlios.de/viewcvs/ial`
– Developer mailing list, `ial-devel@lists.berlios.de`
– User mailing list, `ial-users@lists.berlios.de`

The printed edition of the diploma thesis is supplemented by a CD-ROM which contains snapshots of the Input Abstraction Layer's source code and its Doxygen documentation, among others (Appendix D). An example of the Input Abstraction Layer's compilation and a complete list of the CD-ROM's contents are found in Appendix D, too.

Chapter 8

Conclusion

The creation of the Input Abstraction Layer helps GNU/Linux to fulfill the users' claim for an operating system that just works. Due to its basic approach, the Input Abstraction Layer ensures to report a wide range of different input events. Thus, the functionality of the Input Abstraction Layer affects several fields of application. The achieved goals and the remaining efforts are recapitulated in the following paragraphs.

8.1 Achievements

The Input Abstraction Layer's fundamental idea was—and still is—to close existing gaps between input device drivers and user space applications. The presently implemented modules prove that the Input Abstraction Layer's approach is correct. The Input Abstraction Layer's implementation provides a framework that unifies the input events of the existing input device drivers and enables applications to receive all input events. Due to the numerous bindings offered by D-BUS for all commonly used programming languages, the Input Abstraction Layer is not limiting its usage by redlining applications because of the programming language they are implemented in.

The modular architecture of the Input Abstraction Layer provides two important advantages. On the one hand, the Input Abstraction Layer easily can be adapted to a specific system and the users needs. On the other hand, developers who are missing the support for an input device driver are able to implement a module for the Input Abstraction Layer with little efforts.

Without the Input Abstraction Layer's presence, applications which process user input events have to face two problems. First, applications can not make any assumptions about the availability of input devices on the different systems. Secondly, applications are unsure about the privileges they have at run time. No matter where an input event actually occurred, the Input Abstraction Layer ensures its notification in a unified way. Thus, applications which utilize the Input Abstraction Layer benefit from the unified representation of input events and the output interface which does not require any special permissions to be accessed.

To avoid reaching a dead end, the most important phase of the development was the study of other projects' source code. This study gave hints about how to solve problems in a reasonable way. For several parts of the Input Abstraction Layer, this study kept the development from reinventing the wheel. This phase preceding the actual implementation was important to achieve satisfying results which comply with the original ideas of the project.

Beside the study of source code, it was essential to spot the edge conditions the resulting program will run in. These edge conditions are of particular relevance for the acceptance by both users and developers, especially in the decision regarding the interprocess communication, focused on a fluent integration of the Input Abstraction Layer into GNU/Linux.

8.2 Future Work

The efforts made so far have created a solid groundwork for upcoming challenges. To ensure the propagation of the Input Abstraction Layer and that it goes into action, the proceeding development demands the achievements of several goals. First, it is necessary to implement as many Input Abstraction Layer modules as possible. This approach will result in a widely acceptance as many users derive benefit from the project. The amount of users has a direct impact on the quality of the Input Abstraction Layer. Feedback and user reports are most valuable for leading the development into the right direction—the direction where the users want the project to go. The future development also needs to pay attention to new input devices supported by Linux. It is reasonable to expand the Input Abstraction Layer's functionality to support other input devices than keyboards. For example, special buttons found on mice, trackballs, joysticks, steering wheels and other peripherals.

Applications implementing user interfaces for the abstract input events are required to make actual usage of the Input Abstraction Layer. Such user interfaces have to focus on all users' needs. For example, a console-based user interface is needed as well as user space interfaces for the various graphical desktop environments. This guarantees that users are not discriminated against due to their habits and preferences on how to use GNU/Linux.

The applications implementing user interfaces for the Input Abstraction Layer have to ensure that the users have full control over the action executed upon an input event. For example, the action triggered by an abstract input event can be the execution of an arbitrary application. Other possible actions are functions offered by Linux kernel drivers. The survey (Appendix B, §B.5) revealed that users demand functions such as adjusting the brightness of a laptop's display, changing the volume with labeled, proposed function keys and the possibility to switch between LCD and CRT screen on laptops. It is due to the Input Abstraction Layer that such functions can be mapped to the proposed keys of input devices.

The Input Abstraction Layer's project continues its evolution among the free and open source community. Everyone is free to use it for any purpose, either under the terms of the GPL or the AFL. It is free for personal use as well as for commercial use. Developers are most welcome to participate in the future development of the Input Abstraction Layer. Users will appreciate the project's achievements by enjoying the increase of control over their system. Choosing open source implies choosing freedom.

Appendix A

Utilized Free and Open Source Software

The diploma thesis has been accomplished using free and open source software exclusively. This chapter contains a comprehensive list of the utilized applications and their area of application. Additionally, the licenses of the applications are specified.

A.1 Typesetting

The diploma thesis is typeset with teTeX which is a TeX distribution that consists only of free software. The various parts of teTeX are released under the terms of different licenses:

- GNU General Public License (GPL), [Fsf91]
- GNU Lesser General Public License (LGPL), [Fsf99]
- GNU Free Documentation License (FDL), [Fsf02]

The layout is based upon Peter Wilson's class Memoir. Memoir is a flexible class for typesetting general fiction, non-fiction and mathematical works as books, reports, articles or manuscripts. Additionally, several custom LaTeX macros have been used. The class Memoir is released under the terms of the LaTeX Project Public License (LPPL) [Lat01].

teTeX is available from `http://www.tug.org/teTeX/`. The class Memoir is distributed by the Comprehensive TeX Archive Network (CTAN). It is available from `http://www.ctan.org/tex-archive/macros/latex/contrib/memoir/`.

A.2 Figures

All figures have been either created or reworked using Inkscape which is a Scalable Vector Graphics (SVG) editor. It is a fork of the Sodipodi project. Inkscape offers a reasonable export of SVG to PostScript which is a crucial point to include graphics in a LaTeX document. Inkscape is released under the terms of the GPL and is available from `http://www.inkscape.org`.

A.3 Statistical Calculations

Both the survey (Appendix B) and the latency measurements (Chapter 6, §6.2) have been evaluated using Gnumeric. Gnumeric is a powerful spreadsheet software and offers the possibility to generate graphs. Generated graphs can be exported to SVG. Gnumeric is released under the terms of the GPL and is available from `http://www.gnome.org/projects/`.

A.4 Survey

The implementation of the survey (Appendix B) was realized with the Unit Command Climate Assessment and Survey System (UCCASS), a PHP and MySQL based survey system. It is highly customizable and easy to set up. Gathered data of surveys can be exported to comma separated values (CSV) format and HTML tables. UCCASS is released under the terms of the Affero General Public License (AGPL) [Aff02]. Compared to the GPL, the AGPL has an additional section which covers the use of software over a computer network. The UCCASS source code is available from the project site located at `http://www.bigredspark.com/survey.html`.

Appendix B

Survey: Current State of Linux Input Devices

The survey “Current State of Linux Input Devices” has been completed by 305 participants. The duration of the survey was five days starting on the 13th of October 2004. The evaluation of the survey’s results mentioned below is performed in Chapter 1, §1.1.

B.1 Announcement

The survey was announced sending the following e-mail:

```
Subject: Current State of Linux Input Devices
From: Timo Hoenig <thoenig@suse.de>
Date: Wed, 13 Oct 2004 19:18:15 +0200

Hi,

I am carrying out a survey on the current state of Linux input devices.
The results will have influence on my diploma thesis and the
corresponding project which is called Input Abstraction Layer [1].

The survey is located at: http://ial.berlios.de/survey/

Anyone who uses Linux -- either on a desktop, a laptop or both -- is
more than welcome to participate.

I appreciate every single contribution to the survey.

Best Regards,

   Timo

[1] Input Abstraction Layer
Web: http://developer.berlios.de/projects/ial/
```

SVN: `http://svn.berlios.de/viewcvs/ial/`

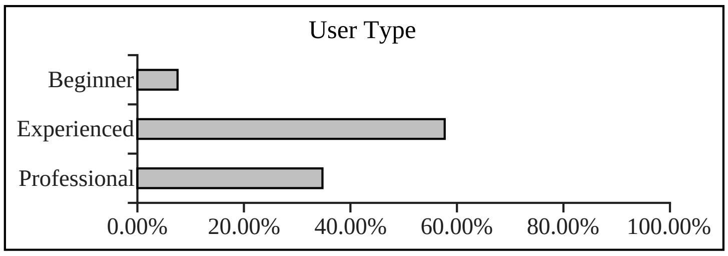
This e-mail was sent to the following mailing lists:

– Linux Laptop, `<linux-laptop@mobilix.org>`
– Debian Laptop, `<debian-laptop@lists.debian.org>`
– Debian User, `<debian-user@lists.debian.org>`
– Fedora Desktop, `<fedora-desktop-list@redhat.com>`
– Gentoo Desktop, `<gentoo-desktop@lists.gentoo.org>`
– Gnome, `<gnome-list@gnome.org>`
– KDE, `<kde-linux@kde.org>`
– Dell Laptop, `<linux-dell-laptops@yahoogroups.com>`
– IBM Laptop, `<linux-thinkpad@linux-thinkpad.org>`
– Sony Laptop, `<linux-sony@insue.com>`
– Toshiba Laptop, `<tlinux-users@linux.toshiba-dme.co.jp>`

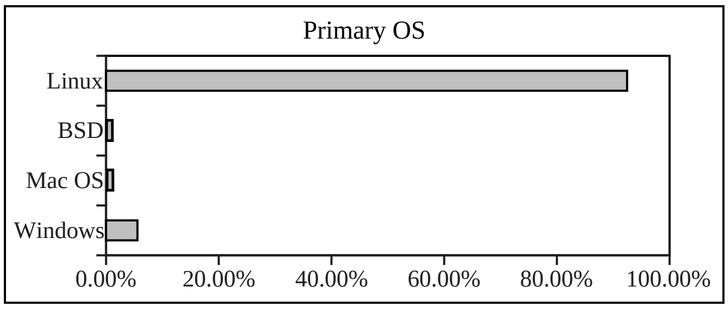
B.2 General

Questions asked in this section are about the participant, the usage of Linux and the used software environment.

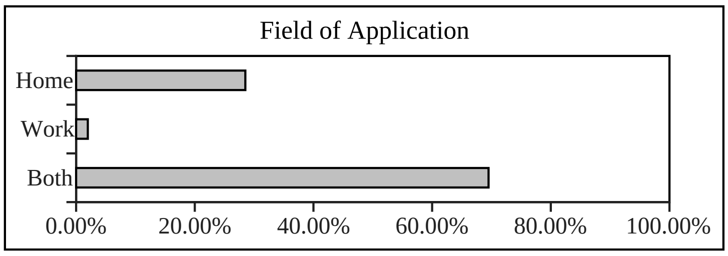
Question 1 (single choice, mandatory): Please classify yourself.



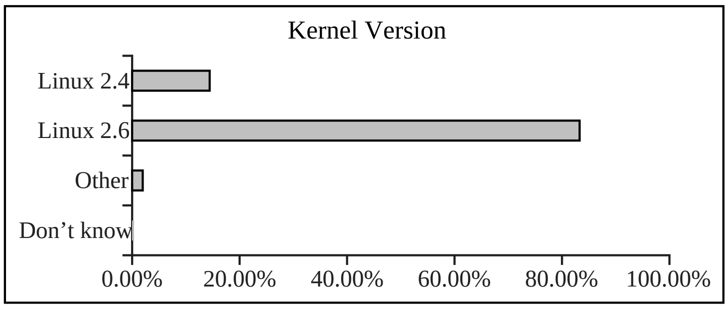
Question 2 (single choice, mandatory): What is your primary desktop operating system?



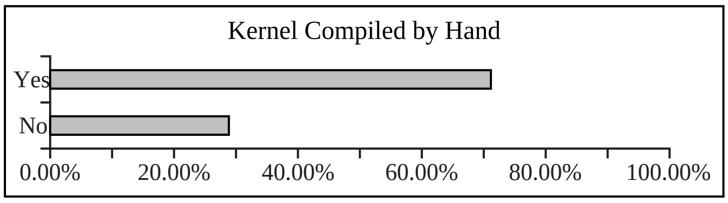
Question 3 (single choice, mandatory): Where do you use Linux?



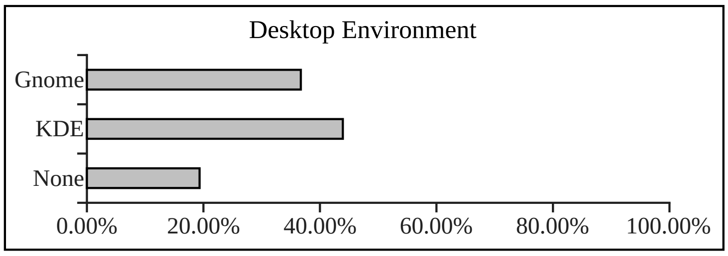
Question 4 (single choice, mandatory): Which kernel are you running?



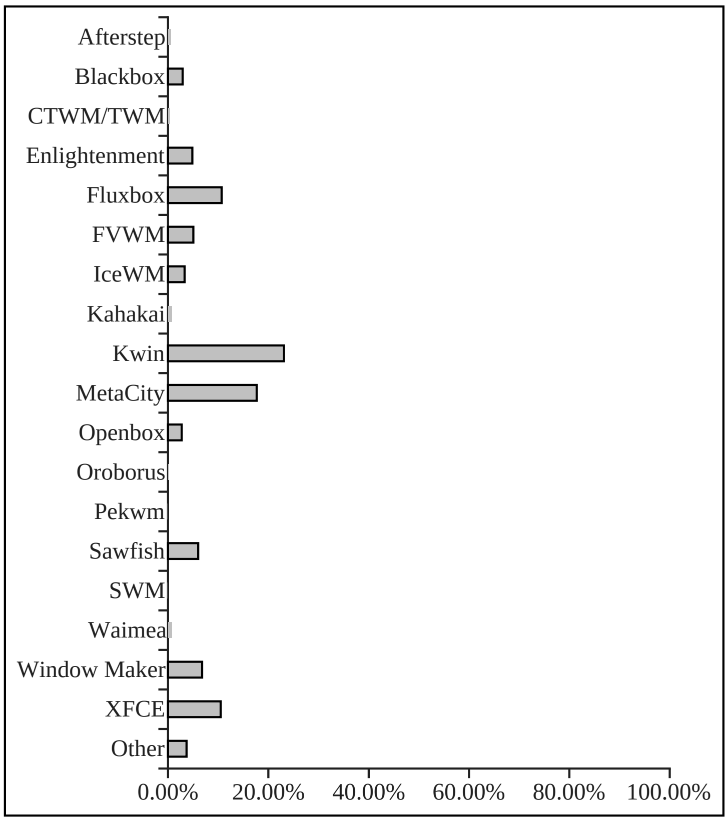
Question 5 (single choice, mandatory): Do you compile your own kernel?



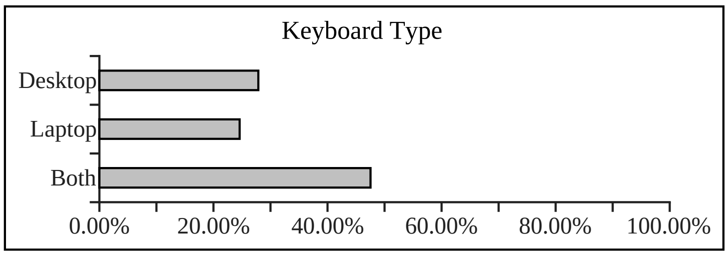
Question 6 (single choice, mandatory): Which desktop environment do you prefer?



Question 7 (multiple choice, mandatory): Which window manager(s) do you prefer?



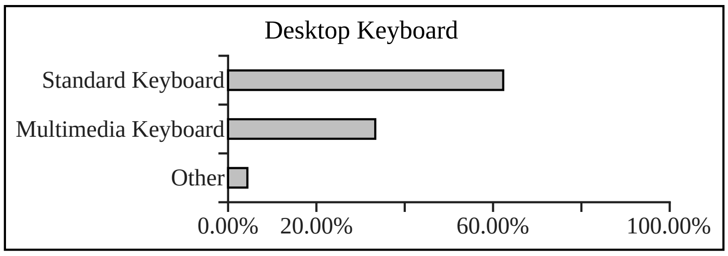
Question 8 (single choice, mandatory): On what kind of system do you run Linux?



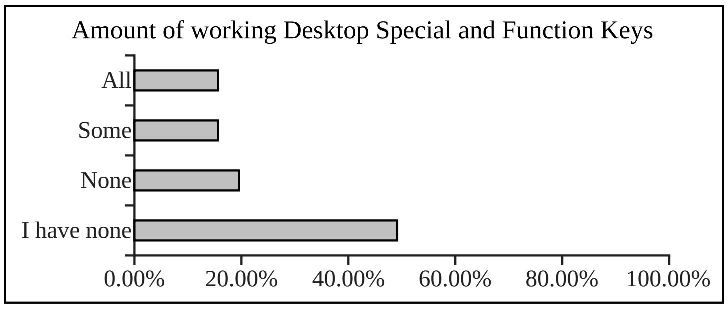
B.3 Desktop Systems

Questions asked in this section are about keyboards used on desktop systems running Linux.

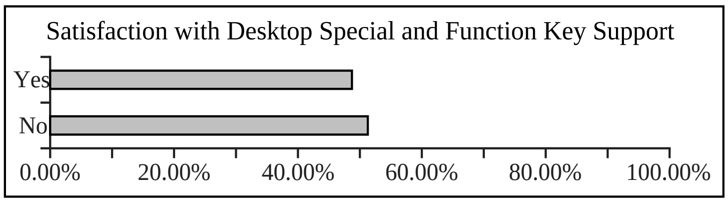
Question 9 (multiple choice, mandatory): What kind of keyboard(s) are you using?



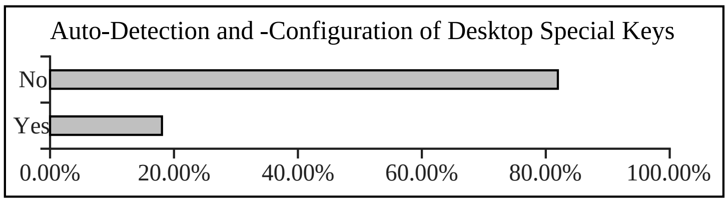
Question 10 (single choice, mandatory): Are the multimedia or special function keys on your keyboard working under Linux?



Question 11 (single choice, mandatory): Are you satisfied with the support of multimedia or special function keys on your keyboard under Linux?



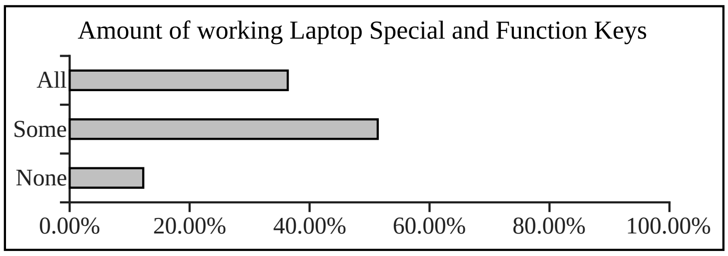
Question 12 (single choice, mandatory): Did your distribution auto-detect those keys?



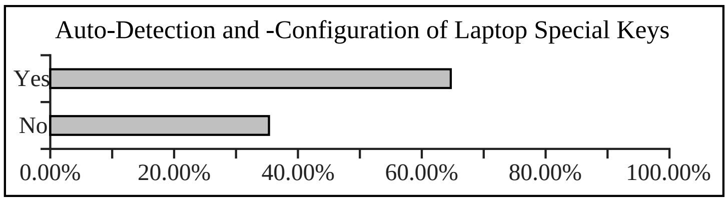
B.4 Laptop Systems

Questions asked in this section are about built-in keyboards used on mobile systems running Linux.

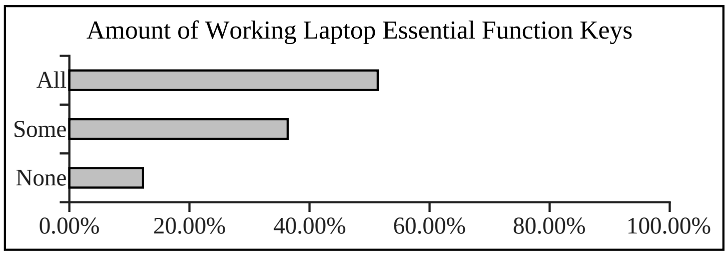
Question 13 (single choice, mandatory): Are the function keys working as supposed?



Question 14 (single choice, optional): If yes, did you need additional software or a kernel driver to get the function keys working?



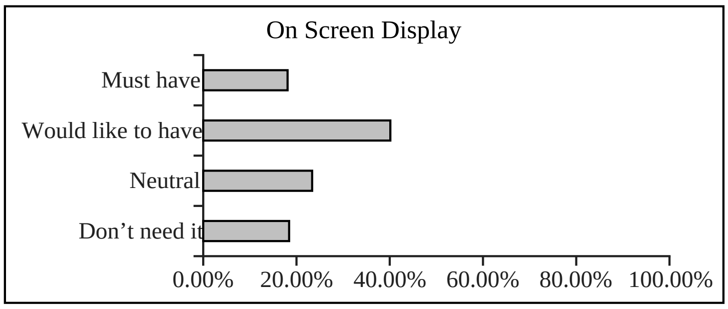
Question 15 (single choice, mandatory): Are essential function keys (e.g. switching CRT/LCD, changing brightness) working?



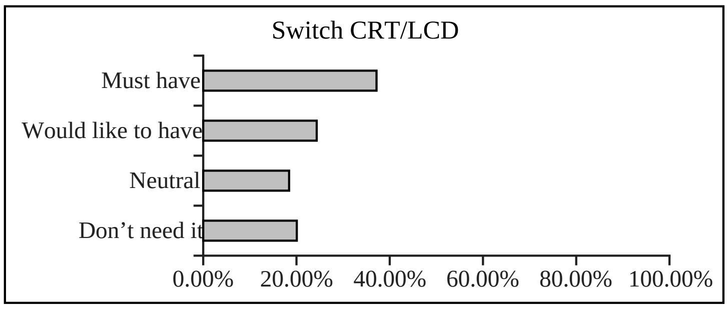
B.5 Demand of Features

Questions asked in this section are about features demanded by the participants of the survey.

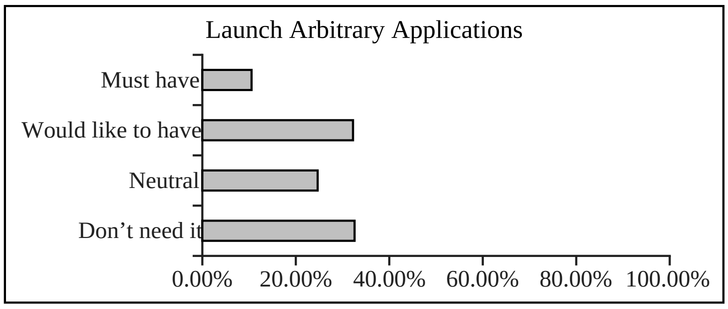
Question 16 (single choice, optional): Would you like to have an on screen display (e.g. for screen brightness, volume)?



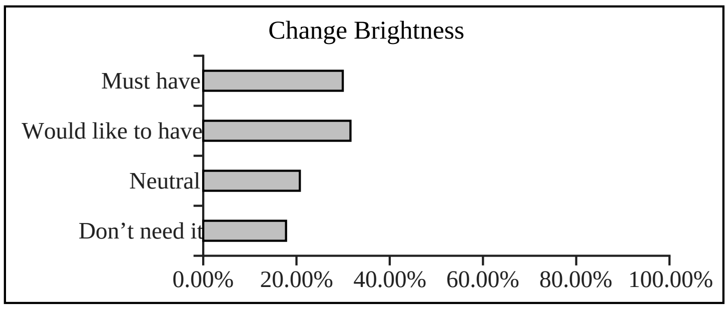
Question 17 (single choice, optional): Would you like to be able to switch between LCD and CRT with the appropriate function key?



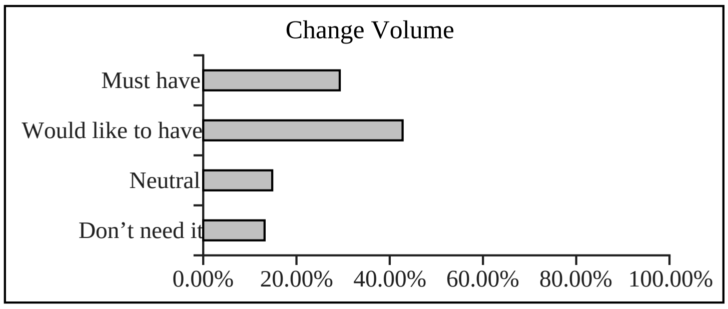
Question 18 (single choice, optional): Would you like to launch arbitrary applications with function/multimedia keys?



Question 19 (single choice, optional): Would you like to be able to change the screen brightness using the appropriate function keys?



Question 20 (single choice, optional): Would you like to be able to change the volume of your sound card using the appropriate function keys?



Appendix C

Linux Function Key Support for Laptops

Chapter 2 discusses several Linux input device driver for mobile computers. The subsequent lists contain the laptop models supported by each driver. These lists are not exhaustive, though they contain all models which are definitely supported.

C.1 Acer System Management Mode Driver

– Acer Aspire 1350, 1450/Ferrari 3000, 1600 and 2000
– Travelmate C110, 210, 220, 230, 290, 350, 360, 370, 520, 610, 620, 630, 650, 660 and 800 series
– Fujitsu Siemens Amilo 7400, 7820 and Pro V200
– Medion MD9783 and MD40100

C.2 Asus/Medion ACPI Driver

– Asus A1340D, A1300F and A2500H
– Asus D1
– Asus L1400B, L2000D, L2000E, L3400D, L3800C, L3H, L4400L, L4500R, L5800C, L8400L and L84F
– Asus M1300A, M2400A, M2400E, M2400N, M3700N, M5200N and M6800N
– Asus S1300A and S1300N
– Asus S200/Victor MP-XP7210 and S5200N
– Medion 9675
– Samsung P30

C.3 Dell System Management Mode Driver

– Inspiron 1100, 2650, 3700, 3800, 4000, 4100, 4150, 5100, 5150, 8000, 8100 and 8200

– Latitude C400, C510, C600, C610, C800, C810, C840, CPiA, CPx J750GT, D600, D800 and X200

C.4 Hewlett Packard OmniBook Driver

– HP OmniBook XE2

– HP OmniBook XE3 GF, GC, GD, GE and compatible

– HP OmniBook XE4500 and compatible

– HP OmniBook 500, 510, 4150, 6000 and 6100

– Fujitsu Amilo D

– Recent Toshiba Satellite laptops which do not have a Toshiba BIOS

C.5 IBM ThinkPad ACPI Driver

– IBM ThinkPad A21e, A22p, A30p, A31 and A31p

– IBM ThinkPad G40

– IBM ThinkPad R32, R40, R40e, R50, R50p and R51

– IBM ThinkPad T20, T21, T22, T23, T30, T40, T40p, T41, T41p, T42 and T42p

– IBM ThinkPad X20, X31 and X40

C.6 IBM ThinkPad (NVRAM)

– IBM ThinkPad A30 and A31

– IBM ThinkPad R30, R31, R32 and R40

– IBM ThinkPad S30

– IBM ThinkPad T20, T23, T30 and T40

– IBM ThinkPad X20, X24, X30, X40

C.7 Panasonic ACPI Driver

– Panasonic R1

– Panasonic R2

– Panasonic R3

– Panasonic T2

– Panasonic W2

– Panasonic Y2

C.8 Toshiba ACPI Driver

- Toshiba Libretto L5W
- Toshiba Portege 2000, 2010, 3440CT, 4000, 7020CT and R100
- Toshiba Satellite 1410-303, 1410-604 and 1800-821
- Toshiba Satellite 2405, 2410-303, 2410-304S, 2410-514, 2450-401 and 2450-S203
- Toshiba Satellite 5005-504, 5100-201, 5200-801, 5200-802, 5200-903, 5202-S503, 5205, 5205-S703 and 5205-S705
- Toshiba Satellite A10-S100, A10-S203, A20-S103, A25-S207 and A40-211
- Toshiba Satellite M30-164 and M30-344
- Toshiba Satellite Pro 490CDT, Pro 6000, Pro 6100 and Pro M10
- Toshiba Tecra 8100 and 9000
- Toshiba Texra M1
- Toshiba Dynabook EX1/524CDET

Note: All recent ACPI-compliant Toshiba laptops which have a Toshiba BIOS are supported by this driver.

C.9 Toshiba System Management Mode Driver

All Toshiba laptops which have a Toshiba BIOS are supported by this driver.

Appendix D

Source Code

The printed edition the diploma thesis is supplemented by a CD-ROM. This CD-ROM contains the source code of the Linux kernel version 2.6.9, the source code of the Input Abstraction Layer and the Input Abstraction Layer Latency kernel driver. Additionally, the diploma thesis is included as source and as PDF file in the directory `/thesis`.

Beside the unpacked source code, all programs are included as archives in the directory `/archives`. These binary files are signed using GNU Privacy Guard (GPG, `http://www.gnupg.org`). The signatures' filenames are equal to the filename of the binaries but with the additional extension `.sign`. The public key `public_key.asc`—which is required for the verification—is stored in the root directory of the CD-ROM. It is available online, too: `http://ial.berlios.de/public_key.asc`. For example, to verify the integrity of the file `ial-20041215.tar.gz` using its signature `ial-20041215.tar.gz.sign`, the following commands need to be executed:

```
$ gpg --import ./public_key.asc
gpg: please see http://www.gnupg.org/faq.html for more information
gpg: key B3054066: public key "Timo Hoenig <thoenig@nouse.net>" imported
gpg: Total number processed: 1
gpg:               imported: 1

$ gpg --verify ial-20041215.tar.gz.sign ial-20041215.tar.gz
gpg: Signature made Tue Dec 14 21:19:40 2004 CET using DSA key ID B3054066
gpg: Good signature from "Timo Hoenig <thoenig@nouse.net>"
gpg:                 aka "Timo Hoenig <thoenig@suse.de>"
```

Since digital versions of the diploma thesis are missing the CD-ROM, the following sections include references where the source code can be obtained on the Internet.

D.1 Linux Kernel

The Linux kernel 2.6.9 is included twice on the CD-ROM. The first version is an unmodified version, the second is a patched version and includes the Input Abstraction Layer Latency driver as well as the patched input device drivers (Chapter 6, §6.2).

CD-ROM:
Linux 2.6.9 (unpacked, unmodified):
`/linux/linux-2.6.9`
Linux 2.6.9 (unpacked, with IAL Latency patch):
`/linux/linux-2.6.9_iallatency`
Linux 2.6.9 (archive, unmodified):
`/archives/linux-2.6.9.tar.gz`

Internet:
Linux 2.6.9 (archive):
`http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.9.tar.gz`

D.2 Input Abstraction Layer

The Input Abstraction Layer's source code can be obtained using Subversion. The following example shows how the latest development branch of the Input Abstraction Layer is obtained:

```
$ svn checkout svn://svn.berlios.de/ial/trunk
A  trunk/mkinstalldirs
A  trunk/Makefile.in
A  trunk/Doxyfile
...
A  trunk/NEWS
A  trunk/aclocal.m4
A  trunk/install-sh
Checked out revision 52.
$ svn info ./trunk
Path: trunk
URL: svn://svn.berlios.de/ial/trunk
Repository UUID: 8e7fc960-65e5-0310-a079-fea189f357c6
Revision: 52
Last Changed Author: thoenig
Last Changed Rev: 52
Last Changed Date: 2004-12-14 20:32:05 +0100 (Tue, 14 Dec 2004)
```

The source code included on the CD-ROM is a snapshot as of the 15th December 2004. Since it is a GNU Autotools package (Chapter 7, §7.2), the following three steps are required to compile and install the Input Abstraction Layer:

```
$ ./configure --prefix=/usr --sysconfdir=/etc
...
$ make
...
$ make install
```

If the source code is obtained using Subversion, it is required to run `./autogen.sh` prior to `./configure`. The last command `make install` needs to be executed with `root` privileges. The Input Abstraction Layer Daemon `iald` is installed to the directory `/usr/sbin/`. The modules are stored in the directory `/usr/lib/ial/modules`. The configuration files are copied to the directories `/etc/iald` and `/etc/dbus-1/system.d`. To run the Input Abstraction Layer daemon, `iald` needs to be executed with `root` privileges. An example client for receiving events from the Input Abstraction Layer is included (`/usr/bin/ialmon`). Prior to the installation of the Input Abstraction Layer, the following software needs to be installed on the system:

– GNU Compiler Collection (`http://gcc.gnu.org/`)
– D-BUS (`http://www.freedesktop.org/Software/dbus`)
– GLib (`http://www.gtk.org/download/`)
– Libxml2 (`http://www.xmlsoft.org/downloads.html`)

CD-ROM:
Input Abstraction Layer (snapshot, unpacked):
`/ial/ial-20041215`
Input Abstraction Layer (snapshot, archive):
`/archives/ial-20041215.tar.gz`
Input Abstraction Layer Documentation (Doxygen, unpacked):
`/ial/doc`

Internet:
Input Abstraction Layer (snapshot, archive):
`http://ial.berlios.de/snapshots/ial-20041215.tar.gz`
Input Abstraction Layer (Subversion repository):
`svn://svn.berlios.de/ial`
Input Abstraction Layer Documentation (Doxygen, unpacked):
`http://ial.berlios.de/doc/`

D.3 Latency Kernel Driver

The Input Abstraction Layer Latency kernel driver is included in the modified Linux 2.6.9 source tree (`linux-2.6.9_iallatency`) and as a separate patch against Linux 2.6.9. This patch can be applied to the Linux kernel as follows:

```
$ patch -p1 -i iallatency_patch/iallatency_2.6.9.patch
patching file drivers/acpi/button.c
patching file drivers/acpi/toshiba_acpi.c
patching file drivers/input/keyboard/atkbd.c
patching file init/Kconfig
patching file init/Makefile
patching file init/iallatency.c
patching file init/main.c
```

The kernel has to be recompiled with the configuration option `INPUT_IALLATENCY` either set to *y* (compile static) or *m* (compile as module). Further information is supplied by the documentation of the Input Abstraction Layer Latency driver. This documentation is found in the file `README` and is part of the driver's package. A user space application for setting `received_jiffies` is included by the Input Abstraction Layer (`/usr/bin/iallatency`).

CD-ROM:
Input Abstraction Layer Latency Driver (unpacked):
`/ial/iallatency_patch/iallatency_2.6.9.patch`
Input Abstraction Layer Latency Driver (archive):
`/archives/iallatency-0.1.tar.gz`

Internet:
Input Abstraction Layer Kernel Driver (archive):
`http://ial.berlios.de/iallatency/`

Appendix E

Copyright

The diploma thesis is licensed under the Creative Commons Attribution License. A summary of the license is given below, followed by the full legal text.

E.1 Commons Deed

Creative Commons – Commons Deed Attribution 2.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must give the original author credit.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code.

E.2 Legal Code

Creative Commons – Legal Code Attribution 2.0

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;
b. to create and reproduce Derivative Works;
c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;
d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.
e. For the avoidance of doubt, where the work is a musical composition:
    i. **Performance Royalties Under Blanket Licenses**. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.
    ii. **Mechanical Rights and Statutory Royalties**. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work (”cover version”) and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).
f. **Webcasting Rights and Statutory Royalties**. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

Abbreviations

ACPI
: Advanced Configuration and Power Interface

AFL
: Academic Free License

AGPL
: Affero General Public License

CLI
: Command Line Interface

CRT
: Cathode Ray Tube

CTAN
: Comprehensive TeX Archive Network

DCOP
: Desktop Communications Protocol

HCI
: Hardware Control Interface

FDL
: GNU Free Documentation License

FIFO
: First In First Out

FSH
: Filesystem Hierarchy Standard

GPG
: GNU Privacy Guard

GPL
: GNU General Public License

GUI
: Graphical User Interface

HID
: Human Interface Device

HIDP
: Human Interface Device Protocol

IPC
: Interprocess Communication

L2CAP
: Logical Link Control and Adaptation Protocol

LCD
: Liquid Crystal Display

LGPL
: GNU Lesser General Public License

LPPL
: LaTeX Project Public License

LSB
: Linux Standard Base

LPPL
: LaTeX Project Public License

NVRAM
: Non-Volatile Random Access Memory

PDF
: Portable Document Format

SCI
: System Control Interface

SMM
: System Management Mode

SVG
: Scalable Vector Graphics

UCCASS
: Unit Command Climate Assessment and Survey System

USB
: Universal Serial Bus

UPS
: Uninterruptible Power Supply

XML
: Extensible Markup Language

Glossary

Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) specification is an open industry standard developed by HP, Intel, Microsoft, Phoenix and Toshiba that defines common interfaces for hardware recognition, motherboard and device configuration and power management [Wik04a] [Hoe04].

Bluetooth

Bluetooth is an industrial specification for wireless personal area networks (PANs) first developed by Ericsson, later formalized by the Bluetooth Special Interest Group (SIG). Bluetooth provides a way to connect and exchange information between devices like personal digital assistants (PDAs), mobile phones, laptops, PCs, printers and digital cameras via a secure, low-cost, globally available short range radio frequency.

Bluetooth lets these devices talk to each other when they come in range, even if they're not in the same room, as long as they are within 10 metres (32 feet of each other) [Wik04e].

Bus

In computer architecture, a bus is a subsystem that transfers data or power between computer components inside a computer or between computers. Unlike a point-to-point connection, a bus can logically connect several peripherals over the same set of wires.

Early computer buses were literally parallel electrical buses with multiple connections, but the term is now used for any physical arrangement that provides the same logical functionality as a parallel electrical bus. Modern computer buses can use both parallel and bit-serial connections, and can be wired in either a multidrop (electrical parallel) or daisy chain topology, or connected by switched hubs, as in the case of USB. [Wik04h]

Daemon

In Unix and other computer operating systems, a daemon–sometimes called a phantom—is a particular class of computer program that runs in the background, rather than under the direct control of a user; they are usually instantiated as processes.

Systems often "launch" daemons at start-up time: they often serve the function of responding to network requests, hardware activity, or other programs by performing some task. Daemons can also configure hardware, run scheduled tasks and perform a variety of other tasks [Wik04g].

Filesystem Hierarchy Standard

The filesystem standard has been designed to be used by Unix distribution developers, package developers, and system implementors. However, it is primarily intended to be a reference and is not a tutorial on how to manage a Unix filesystem or directory hierarchy [Fsh04].

Hotplug

Hotplug lets you plug in new devices and use them immediately. That means that users won't need to learn so much system administration; systems will at least partially autoconfigure themselves. Initially, hotplug included support for USB and PCI (Cardbus) devices, and could automatically configure some common network interfaces. Updated versions include IEEE 1394 (Firewire/i.Link) support and can download firmware to USB devices that need it. On mainframes, S/390 channel devices use hotplugging to report device attach and other state change events. For laptops, newer kernels also include support for reporting docking station activity.

In the Linux 2.6 kernel, hotplugging has been integrated with the driver model core so that any bus or class can report hotplug events when devices are added or removed (...) There's work afoot to improve the situations for hotplugging many kinds of devices, including things like disks, power supplies (many newer UPSes are USB-programmable), input devices, and even more [Lhp04].

Linux

Linux is the name of a computer operating system and its kernel. It is the most famous example of free software and of open-source development [Wik04b].

In computing, the Linux kernel is a free Unix-like operating system kernel created by Linus Torvalds in 1991 and subsequently improved with the assistance of developers around the world. It was originally developed for the Intel 80386 processor but has since been ported to many other platforms. It is written almost entirely in C with some GNU C language extensions and AT&T assembly language. Developed under the GNU General Public License, the source code for Linux is free software. The kernel is best known as the core of Linux operating systems. Distributions of software based on this kernel are called Linux distributions [Wik04c].

Linux Standard Base

The Linux Standard Base, acronym form LSB, is a joint project by several Linux distributions under the organizational structure of The Free Standards Group to lay out and standardize the internal structure of Linux-based operating systems. The LSB is based on the POSIX specification, the Single Unix Specification, and several other open standards, but extends them in certain areas [Wik04f].

Universal Serial Bus

The Universal Serial Bus (USB) provides a serial bus standard for connecting devices, usually to a computer, but it also is in use on other devices such as set-top boxes, game consoles and PDAs. A USB system has an asymmetric design, consisting of a single host and multiple devices connected in a tree-like fashion using special hub devices. Up to 127 devices may be connected to a single host, but the count must include the hub devices as well, so the total useful number of connected devices diminishes somewhat [Wik04d].

XML

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [W3c04].

Bibliography

Print Publications

[CSF04] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato. *Version Control with Subversion.* O'Reilly. June 2004. (ISBN 0-596-00448-6), 2004.
Available free of charge from `http://svnbook.red-bean.com/`.

[FB03] Karl Fogel, Moshe Bar. *Open Source Development with CVS.* Paraglyph Press. July 2003. (ISBN 1-932111-81-6), 2003.
Available free of charge from `http://cvsbook.red-bean.com/`.

[Hoe04] Timo Hönig. *ACPI Implementation in Linux 2.6: The Small Sleeper.* Linux Magazine. Issue #40, March 2004. (ISSN 14715678), 2004.

[Lau04] Andrew M. St. Laurent. *Understanding Open Source and Free Software Licensing.* O'Reilly. August 2004. (ISBN 0-596-00581-4), 2004.

[Lov03] Robert Love. *Linux Kernel Development.* Sams. September 2003. (ISBN 0-672-32512-8), 2003.

[Mau03] Wolfgang Mauerer. *Linux Kernelarchitektur.* Hanser Fachbuchverlag. November 2003. (ISBN 3-446-22566-8), 2003.

[Ray01] Eric S. Raymond. *The Cathedral and the Bazaar.* O'Reilly. February 2001. (ISBN 0-596-00108-8), 2001.
Available free of charge from `http://www.catb.org/~esr/writings/cathedral-bazaar/`.

[Spi03] Diomidis Spinellis. *Code Reading: The Open Source Perspective.* Addison-Wesley. May 2003. (ISBN 0-201-79940-5), 2003.

[Ste92] W. Richard Stevens. *Advanced Programming in the Unix Environment.* Addison-Wesley. June 1992. (ISBN 0-201-56317-7), 1992.

[Ste98] W. Richard Stevens. *UNIX Network Programming, Volume 2: Interprocess Communications.* Prentice Hall. August 1998. (ISBN 0-13-081081-9), 1998.

[VET00] Gary V. Vaughan, Ben Elliston, Tom Tromey, Ian Lance Tayl. *GNU Autoconf, Automake, and Libtool.* O'Reilly. October 2000. (ISBN 1-57870-190-2), 2000.
Available free of charge from `http://sources.redhat.com/autobook/`.

Online Ressources

[Acp04] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. *Advanced Configuration and Power Interface Specification v3.0.* September 2004. (Available from `http://www.acpi.info/spec.htm`)

[Aff02] Affero Inc. *Affero General Public License.* March 2002. (Available from `http://www.affero.org/oagpl.html`)

[Fsh04] Filesystem Hierarchy Standard Group. *Filesystem Hierarchy Standard v2.3.* 2004. (Available from `http://www.pathname.com/fhs/pub/fhs-2.3.pdf`)

[Fsf91] Free Software Foundation. *GNU General Public License.* June 1991. (Available from `http://www.fsf.org/licenses/gpl.txt`)

[Fsf99] Free Software Foundation. *GNU Lesser General Public License.* February 1999. (Available from `http://www.fsf.org/licenses/lgpl.txt`)

[Fsf02] Free Software Foundation. *GNU Free Documentation License.* November 2002. (Available from `http://www.fsf.org/licenses/fdl.txt`)

[Fsf04a] Free Software Foundation. *GNU Coding Standards.* December 2004. (Available from `http://www.fsf.org/prep/standards/`)

[Fsf04b] Free Software Foundation. *Various Licenses and Comments about Them.* November 2004. (Available from `http://www.fsf.org/licenses/license-list.html`)

[Fsg04] Free Standards Group. *Linux Standard Base Specification v2.0.* 2004. (Available from `http://refspecs.freestandards.org/LSB_2.0.0/`)

[Iee04] The IEEE and The Open Group. *The Open Group Base Specifications, IEEE Std 1003.1.* 2004. (Available from `http://www.opengroup.org/onlinepubs/009695399/`)

[Lat01] LaTeX3 Project. *The LaTeX Project Public License.* December 2001. (Available from `http://www.latex-project.org/lppl/lppl-1-3.txt`)

[Lhp04] Linux Hotplug Project. *Linux Hotplugging.* October 2004. (Available from `http://linux-hotplug.sourceforge.net/`)

[Low04] Kwan Lowe. *Kernel Rebuild Guide.* 2004. (Available from `http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html`)

[OSS04] Open Source Initiative. *The Approved Licenses.* 2004. (Available from `http://www.opensource.org/licenses/`)

[RG03] Craig Ranta, Steve McGowan. *Bluetooth HID Specification v1.0.* May 2003. (Available from `http://www.bluetooth.org/foundry/adopters/document/HID_Spec_v1_0/en/3/HID_Spec_v1_0.zip`)

[Ros04] Lawrence E. Rosen. *The Academic Free License v. 2.1.* Mai 2004. (Available from `http://www.rosenlaw.com/afl21.htm`)

[Sig04] Bluetooth Special Interest Group (SIG). *Bluetooth Specification Documents.* October 2004. (Available from `http://www.bluetooth.org/spec/` Note: Registration required.)

[Usb01] USB Implementers Forum. *Device Class Definition for Human Interface Devices (HID).* June 2001. (Available from `http://www.usb.org/developers/devclass_docs/HID1_11.pdf`)

[Wik04a] Wikipedia. *Advanced Configuration and Power Interface.* October 2004. (Available from `http://en.wikipedia.org/wiki/ACPI`)

[Wik04b] Wikipedia. *Linux.* October 2004. (Available from `http://en.wikipedia.org/wiki/Linux`)

[Wik04c] Wikipedia. *Linux Kernel.* October 2004. (Available from `http://en.wikipedia.org/wiki/Linux_kernel`)

[Wik04d] Wikipedia. *Universal Serial Bus.* October 2004. (Available from `http://en.wikipedia.org/wiki/Usb`)

[Wik04e] Wikipedia. *Bluetooth.* October 2004. (Available from `http://en.wikipedia.org/wiki/Bluetooth`)

[Wik04f] Wikipedia. *Linux Standard Base.* November 2004. (Available from `http://en.wikipedia.org/wiki/Linux_standard_base`)

[Wik04g] Wikipedia. *Daemon.* November 2004. (Available from `http://en.wikipedia.org/wiki/Daemon_(computer_software)`)

[Wik04h] Wikipedia. *Bus.* November 2004. (Available from `http://en.wikipedia.org/wiki/Computer_bus`)

[W3c04] World Wide Web Consortium (W3C). *Extensible Markup Language (XML).* November 2004. (Available from `http://www.w3.org/xml/`)

All links verified: 9th January 2005

Index