The use of Windows USB scanner drivers in Linux

Igor Farinič

2007

The use of Windows USB scanner drivers in Linux

MASTER THESIS

Igor Farinič

COMENIUS UNIVERSITY BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE

Specialization: Informatics

Thesis advisor: RNDr. Jaroslav Janáček

BRATISLAVA 2007

By this I declare that I wrote this diploma thesis by myself, only with the help of the referenced literature and internet resources, under the supervision of my thesis advisor.

Bratislava, 2007 Igor Farinič

Abstract

The thesis is a study of creating environment around Windows USB scanner drivers in Linux operating system. We achieved using Windows USB scanner drivers to drive the scanners in Linux operating system. Also we fully managed to incorporate implemented Windows scanner architecture into Linux SANE scanning architecture.

The thesis starts with a briefing of a Windows driver model and Windows scanning architecture. Then it continues with a Linux driver model and user space implementation of Win32 API in Linux environment called Wine.

The core of the work is a description of our design and design decisions for implemented Windows scanning architecture in Linux environment. Finally, we describe, in details, incorporation of implemented Windows scanning architecture into Linux scanning architecture.

We also provide how to put all the implemented pieces of code together and to setup a scanner to scan images in Linux environment with Windows driver.

Keywords: driver, scanner, Linux, Windows

Preface

Wide spreading of Linux on desktop machines is inhibited by the lack of support for device drivers. Overwhelming majority of device vendors ship their products with drivers for Microsoft Windows platform. Another issue with a lot of devices is that their specification is proprietary, hence it is tedious, sometimes even impossible, to write a Linux driver for a specific device.

Our goal was to use existing Windows drivers and create environment around them that will resemble original Microsoft Windows environment to let the driver drive the device.

Typographic Conventions

Typographic conventions used in this thesis are simple and intuitive:

- `Typewriter`-like font denotes source code, `function()` definitions and related things.
- Files and directories like `[usbscan.c]` for Linux kernel driver are enclosed in square brackets.

Copyrights

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is one large group of devices that connect through Universal Serial Bus. These devices are grouped into classes where each class depends on different Windows subsystem. For example USB printers depend on Microsoft Windows printing subsystem, USB scanners depend on Microsoft Windows still image subsystem. Another difference between these driver classes is that drivers from one class can run solely in user-space mode and thus depend only on Win32 API another class's drivers can run partially in kernel-mode or completely in kernel-mode, thus they also depend on Microsoft Windows Executive.

To achieve our goal we reduced our domain of interest only to USB scanner devices. Drivers for these devices run completely in user-space mode and communicate with kernel through USB scanner driver that runs completely in the kernel.

The reader is expected to have some preliminary knowledge about operating systems.

1.1 Thesis Content

The first several chapters are necessary to understand the whole background around drivers. Chapter 2 describes basics about the file format for all Microsoft's drivers. Information provided in this chapter can be fully utilized during debugging of Windows drivers, but basic understanding of this chapter can help in chapters 4, 6 and 9.

Principles of Universal serial bus functionality can be found in chapter 3.

Detailed description for some aspects of Microsoft Windows driver model can be found in 4. This chapter is dedicated to driver installation, its loading and initialization and principles of device access from drivers. Knowledge from this chapter will be useful in chapter 9, especially in part about implementation of USB scanner kernel module and methods how to implement interface between USB scanner module and user-space scanner driver.

Understanding of Linux driver model in chapter 5 will be presented on the USB subsystem. Its understanding is a condition for implementation of interface for USB scanner module.

Userspace part of Windows drivers for USB scanners use only Win32 API that is implemented in Linux by Wine project. We used this implementation with small modifications to create a wrapper around scanner drivers. First we need to understand architecture and some implementation decisions of Wine to adjust it to our requirements. These are described in chapter 6.

Low level architecture of Windows USB scanner drivers is described in chapter 7. High level API above STI is called TWAIN. As most of the modern scanner drivers conform to TWAIN architecture, we had to modify TWAIN support library implemented in Wine, too. The architecture of TWAIN model is described in chapter 8.

All the information is put together in chapter 9, where the whole implemented architecture is described as a whole. Here we also describe decisions and restrictions that were made during the development phase.

To fully integrate our implemented Windows scanning architecture into Linux environment we decided to incorporate it into SANE architecture. The achievement is described in chapter 10.

By following steps in chapter 11 the reader is welcome to try to setup the scanner in Linux. It is a complete know-how of getting the scanner working with our implementation. The reader is required to have some basic understanding of Linux environment and programming in C language.

Chapter 2

Microsoft Portable Executable

Microsoft Portable Executable is a file format for executables, object files and dynamic link libraries (DLLs) used in 32-bit and 64-bit (with extensions called PE32+) versions of Windows. "Portable" refers to the file format portability across all 32-bit and 64-bit Windows operating systems on all supported CPU types. On NT operating systems the PE format is used for EXE, DLL, OBJ, SYS file types.

Distinction between EXE and DLL files is in its semantics. They both use the same PE file format. The only difference is in one bit that indicates if the file should be treated as EXE or DLL.

2.1 Overview

The most important feature of the PE file format is that the file on disk is very similar to what the image will look like after Windows has loaded. The loader does not need to work hard to create process from the disk file.

All the memory used by the image for code, data, resources, import tables, export tables and other required image data structures is in one contiguous block of memory. All we need to know is where the loader mapped the file into memory. Then we can find all the various pieces of the image by following pointers that are stored as part of the image as shown in figure 2.1.

When file is loaded it is mapped into its virtual address space. Many fields in PE files are located by Relative virtual address (RVA). An RVA is simply the offset of some item, relative to where the file is memory-mapped. The base address is the starting address of a memory-mapped EXE or DLL. We can compute RVA:

$$RVA = VirtualAddress - BaseAddress$$

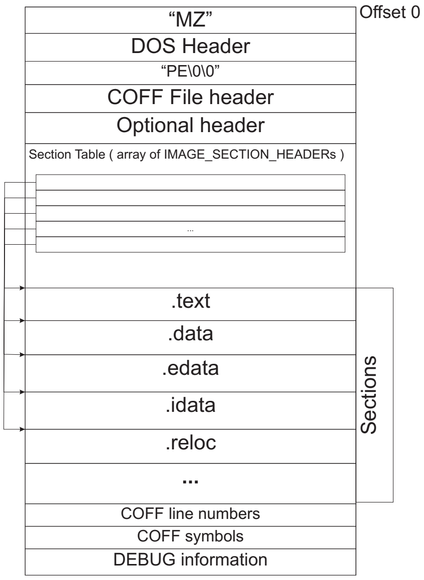

Figure 2.1: The PE file format

The final concept of PE files is sections. Sections contain either code or data. They are blocks of contiguous memory with no size constraints. Some of them contain code or data that program declared and uses directly, while others are created by linker and contain information vital to the operating system.

2.2 File Headers

The PE file has a collection of fields at a known (or easy to find) location that define how the rest of the file looks like. This header contains information about locations and sizes of the code and data areas, what operating system the file is intended for, the initial stack size and other vital information.

The first bytes of the typical PE file are taken up by the Microsoft MS-DOS stub. The MS-DOS stub is a valid application that runs under MS-DOS. The linker places a default stub here, which prints out the message "This program cannot be run in DOS mode" when the image is run in MS-DOS.

After the MS-DOS stub, at the file offset specified at offset 0x3c, is a 4-byte signature that identifies the file as a PE format image file. This signature is "PE\0\0".

Following the PE signature in the PE header are structures: COFF file header and an optional header. In both cases, the file headers are followed immediately by section headers. The COFF file header contains only the most basic information about the file: CPU that this file is intended for, number of sections, timestamp of file creation by linker, pointer to symbol table, number of symbols in the COFF symbol table, size of the optional header and characteristics for the file.

The optional header is optional in the sense that some files (specifically, object files) do not have it. For image files, this header is required.

Not all of the COFF optional header fields are necessarily to be aware of. The most important ones to be aware of are the ImageBase and the Subsystem fields. Details can be found in [1]. Fields for this header are divided into three groups:

Optional Header Standard Fields

The first eight fields of the optional header are standard fields that are defined for every implementation of COFF. These fields contain general information that is useful for loading and running an executable file. They are unchanged for the PE32+ format.

Optional Header Windows-specific Fields

The next 21 fields are an extension to the COFF optional header format. They contain additional information that is required by the linker and loader in Windows.

2.3 Section Table

This table immediately follows the optional header, if any. This positioning is required because the file header does not contain a direct pointer to the section table. Instead, the location of the section table is determined by calculating the location of the first byte after the headers. Make sure to use the size of the optional header as specified in the file header.

Each row of the section table is, in effect, a section header. This section header contains a set of attributes including whether section contains code, if it is read-only or read/write section and whether the data in the section are shared between all processes using the executable. The number of entries in the section table is given by the `NumberOfSections` field in the file header. Beginning address (RVA) of the section can be retrieved from section's record in section table. Sections are ordered by their starting RVAs.

Section represents code or data of some type. The code or data is related in some way. While code is just code, there are multiple types of data. Besides read/write program data, there are other types of data in sections including API import and export tables, resources and relocations.

2.3.1 Common Sections

The following list of sections is not complete. It will include sections that are present almost in every EXE or DLL file. For complete list of sections and for details see [1].

The .text section

The .text section is where all general purpose code emitted by the compiler or assembler ends up. The linker concatenates all the .text section from all OBJs files into one big .text section.

The .data section

The .data section is where all initialized data goes. These data consist of global and static variables that are initialized at the compile time. They also include the string literals. The linker combines all the .data sections from the OBJ files into one .data section.

The .edata section

The .edata section is a list of the functions and data that the PE file exports for other modules. This section can be seen mostly in DLL files and is illustrated on figure 2.2.

At the start of an .edata section is an Image Export Directory structure. This structure is immediately followed by data. There are pointers in the structure to these data. The primary components of an .edata section are tables of function names, entry point addresses and export ordinal values. These arrays are parallel to each other. To find information about a specific function it is needed to look it up in all three arrays.

Note that in many cases there are two functions exported that only differ by one character at the end of the name. This is how UNICODE support is implemented transparently. The functions that end with A are the ASCII compatible functions, while those ending with W are the UNICODE version of the functions. In code, we do not explicitly specify which function to call. Instead the appropriate function is selected via preprocessor.

The .idata section

The .idata section contains information necessary for the loader to determine the addresses of the functions imported from other DLLs and patch them into the executable image. The .idata section is in fact an import table as shown in figure 2.3. In

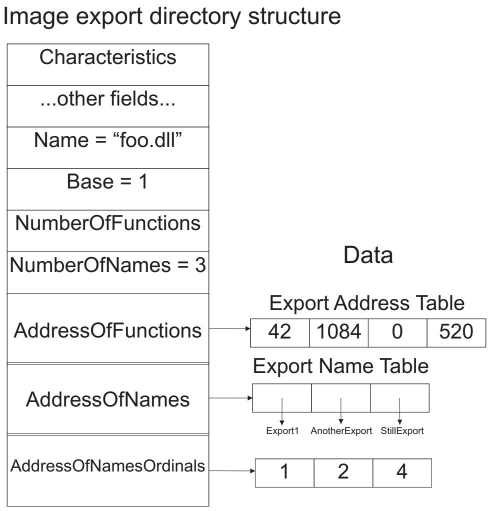

Figure 2.2: The PE export directory

this table there is one entry (of type `IMAGE_IMPORT_DESCRIPTOR`) for each imported DLL that the PE file implicitly links to. Last entry of the table is always filled with `NULL`s to indicate the end of the table.

The important parts of each table entry are the imported DLL name and the two arrays of pointers. These two arrays run parallel to each other and are terminated by a `NULL` pointer entry at the end of each array. The pointers in both arrays point to an `IMAGE_IMPORT_BY_NAME` structure. These arrays' names are Import Address Table (IAT) and Import Name Table (INT). During binding, the entries in the Import Address Table are overwritten with the addresses of the symbols that are being imported. These addresses are the actual memory addresses of the symbols, although technically they are still called virtual addresses. The loader typically processes the binding.

The .rsrc section

The .rsrc section contains all the resources for the image. Resources are indexed by a multiple-level binary-sorted tree structure. For details about how this structure looks like and how to search it see [1].

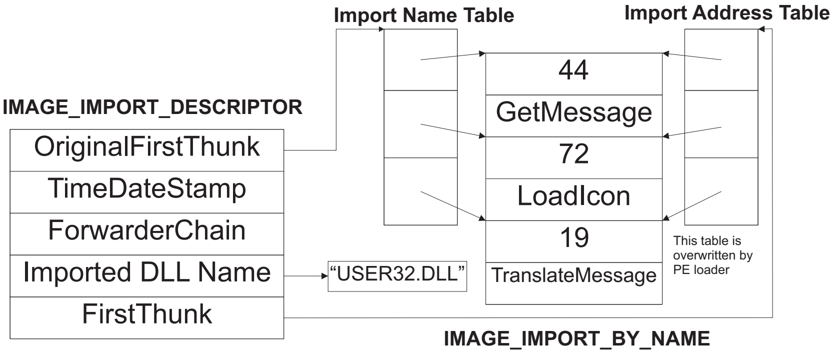

Figure 2.3: The PE Import table descriptor

The .reloc section

The .reloc section holds a table of the base relocations. A base relocation is an adjustment to an instruction or initialized variable value that is needed if the loader could not load the file where the linker assumed it would. If the loader is able to load the image at the linker's preferred base address, then the loader completely ignores the relocation information in this section.

The .tls section

The .tls section refers to thread local storage. For .tls section the memory manager sets up page tables so that whenever a process switches threads, a new set of physical memory pages is mapped to the .tls section's address space. This permits per-thread global variables.

The .pdata section

The .pdata section contains an array of function table entries that are used for exception handling. It is pointed to by the exception table entry in the image data directory. The entries must be sorted according to the function addresses (the first field in each structure) before being emitted into the final image.

2.4 Loading Procedure

The loader is responsible for loading PE file and preparing it in memory for execution. First it will find free virtual address space to map the file in memory. It tries to load the image at the preferred base address. If it is not possible then it will try to load it at different base address.

After this is done, the loader maps the sections in memory. The loader goes through the section table and maps each section at the address calculated by adding the RVA of the section to the base address.

After mapping the section in memory, the loader performs based relocation if the base address is not equal to the preferred base address. Then the Import Table is checked and the required DLLs are loaded. The same procedure for loading executable-mapping sections, based relocations, resolving imports is also applied while loading a DLL. After loading each DLL, the Import Address Table (IAT) is fixed to point to the actual imported function address.

Chapter 3

The Universal Serial Bus

3.1 USB Basics

The Universal Serial Bus is a type of bus that is topologically laid out as a tree built out of several point-to-point links.

A hub is a USB device which extends the number of ports to connect other USB devices. Normally ports of the USB host controller are handled by a virtual root hub. This hub is simulated by the host controller's device driver and helps to unify the bus topology.

The USB host controller is in charge of asking every USB device if it has any data to send. USB device can never start sending data without first being asked to by the host controller. The communication on the USB is done in two directions. Data directed from the host to a device is called downstream or OUT transfer, the other direction is called upstream or IN transfer.

There can be used three different transfer types for communication:

- Control transfers - are used to request and send reliable short data packets. It is used to configure devices and each one is required to support a minimum set of control commands.
- Bulk transfers - are used to request or send reliable data packets up to the full bus bandwidth. Scanners are using this transfer type.
- Interrupt transfers - are similar to bulk transfers which are polled periodically.
- Isochronous transfers - send or receive data streams in real-time with guaranteed bus bandwidth but without reliability.

System communicates with scanner devices only by control and bulk transfers.

3.1.1 Endpoints

The USB communication is realized through endpoints. A USB endpoint can carry data in one direction only, either from the host to the device or from the device to the host.

3.1.2 Interfaces

USB endpoints are bundled up into interfaces. Interface has a zero or more endpoints. USB interfaces handle one type of a USB logical connection only. Some USB devices have multiple interfaces. Because the USB interface represents basic functionality, each USB driver controls an interface. Hence if the device has more than one interface, every interface can be driven by its own driver.

USB interfaces may have alternate settings, which are different choices for parameters of the interface. The initial state of an interface is in the first setting, numbered 0. Alternate settings can be used to control individual endpoints in different ways.

3.1.3 Configurations

USB interfaces are themselves bundled up into configurations. A USB device can have multiple configurations and might switch between them in order to change the state of the device. A single configuration can only be enabled at one point in time. The overview of USB device is illustrated in figure 3.1.

3.2 USB URBs

The USB code communicates with all USB devices using structure URB (USB Request Block). An URB is used to send or receive data to or from a specific USB endpoint on a specific USB device in an asynchronous manner. A USB device driver may allocate many URBs for a single endpoint or may reuse a single URB for many different endpoints. Every endpoint in a device can handle a queue of URBs, so that multiple URBs can be sent to the same endpoint before the queue is full.

The lifecycle of a URB is as follows:

- Created by a USB device driver
- Assigned to a specific endpoint of a specific USB device
- Submitted to the USB core, by the USB device driver

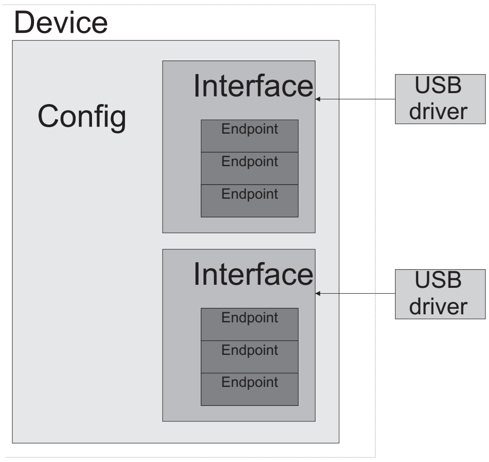

Figure 3.1: USB device overview

- Submitted to the specific USB host controller driver for the specified device by the USB core
- Processed by the USB host controller driver that makes a USB transfer to the device
- When the URB is completed, the USB host controller driver notifies the USB device driver

URBs can also be cancelled any time by the driver that submitted the URB or by the USB core. [1]

---

[1] Corbet, J. et al.: Linux Device Drivers Third Edition, O'Reilly, 2005, p. 335

Chapter 4

Windows Device Drivers

4.1 Type of Windows Drivers

There are two basic kinds of Windows drivers:

- User-mode drivers - are represented by typical user-mode libraries.
- Kernel-mode drivers - run as a part of the Executive, which consists of kernel operating system components (manages I/O, memory, processes, threads, ...). They are implemented as modular components with a defined set of required functionality. All WDM drivers run in kernel mode.

4.1.1 Windows Driver Model (WDM)

Microsoft introduced Windows Driver Model to allow writing of kernel mode drivers that are source-code compatible across all Windows operating systems. All WDM drivers must follow defined WDM rules.

Types of WDM Drivers

There are three types of WDM drivers:

- Bus drivers - drive an I/O logical or physical bus. A bus driver is responsible for detecting and informing PnP manager of devices attached to the bus it controls and also managing the power setting on the bus.
- Function drivers - drive individual device. Bus drivers present devices to function drivers via the PnP manager. The function driver is the driver that exports the operational interface of the device to the operating system. In general, it is the driver with the most knowledge about the operation of the device.

- Filter drivers - filter I/O requests for a device, class of devices, or a bus. They create logical layer above or below function drivers, augmenting or changing the behavior of a device or another driver.

Each device typically has a bus driver for the parent I/O bus, a function driver for the device and zero or more filter drivers for the device. A driver design that requires many filter drivers does not yield optimal performance.

The figure 4.1 shows the relationship between the bus driver, function driver and filter drivers for a device.

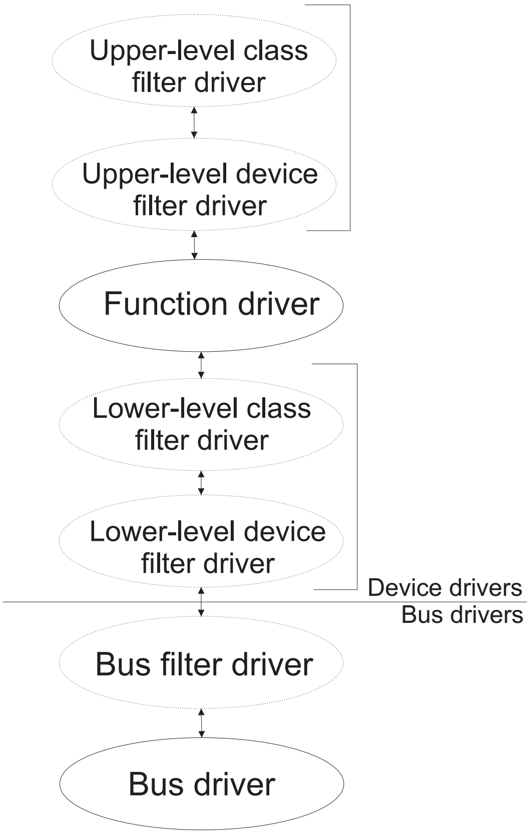
Upper-level class filter driver
Upper-level device filter driver
Function driver
Lower-level class filter driver
Lower-level device filter driver
Device drivers
Bus drivers
Bus filter driver
Bus driver

Figure 4.1: Driver layers

Bus Drivers

A bus driver drives bus controller, adapter or bridge. There is one bus driver for each type of a bus on a machine. A bus driver can service more than one bus, if there is more than one bus of the same type on the machine. The primary responsibilities of a bus driver are to:

- Enumerate the devices on its bus
- Respond to Plug and Play IRPs and power management IRPs

- Multiplex access to the bus
- Generically administer the devices on its bus

During enumeration the bus driver identifies the devices on its bus and creates device objects for them. This approach is described in section 4.2. The method the bus driver uses to identify connected devices depends on the particular bus.

The bus driver does not handle read and write requests for the devices on its bus. Read and write requests to a device are handled by the device's function driver.

A bus driver acts as the function driver for its controller, adapter or bridge and therefore manages device power policy for these components.

Function Drivers

A function driver is the main driver for a device. A function driver is typically written by the device vendor and is required. The PnP Manager loads at most one function driver for a device. A function driver can service one or more devices.

A function driver provides the operational interface for its device. Typically the function driver handles reads and writes to the device and manages device power policy.

The function driver for a device can be implemented as a driver/minidriver pair or class/miniclass driver pair. In such driver pairs, the minidriver is linked to the second driver which is a DLL.

Filter Drivers

Filter drivers are optional drivers that add value to or modify the behavior of a device. A filter driver can service one or more devices.

Bus filter drivers typically add value to a bus. A bus filter driver could, for example, implement proprietary enhancements to standard bus hardware.

Lower-Level Filter Drivers

Lower-level filter drivers typically modify the behavior of device hardware. A lower-level device filter driver monitors and/or modifies I/O requests to a particular device. For example such filters redefine hardware behavior to match expected specifications. A lower-level class filters driver monitors and/or modifies I/O requests for a class of devices. For example, a lower-level class filter driver for mouse devices could provide acceleration.

Upper-Level Filter Drivers

Upper-level filter drivers typically provide added-value features for a device. An upper-level device filter driver adds value for a particular device. For example, an upper-level device filter driver for a keyboard could enforce additional security checks. An upper-level class filter driver adds value for all devices of a particular class. [2]

4.2 Driver Objects and Device Objects

A driver object represents an individual driver in the system. The I/O manager obtains the address of each of the driver dispatch routines from the driver object.

A device object represents a physical or logical device on the system and describes its characteristics.

The I/O manager creates a driver object when a driver is loaded into the system and it then calls the driver's initialization routine, which fills in the object attributes with the driver's entry points.

After loading, a driver can create a device object for each device it controls. The device object represents the device to the driver. Every kernel-mode driver creates at least one device object. Some drivers must create more than one device object. Each driver object represents a single driver that can create one or more device objects. The I/O Manager maintains a list of device objects created by each driver and stores a pointer to this list in the driver object. Each standard driver routine that is passed an IRP is also passed a pointer to the target device object for the I/O request. Most drivers use the device extension of the target device object to maintain device state information or driver-specified context data for the I/O request.

When a driver creates a device object, the driver can optionally assign a name to the device. A name places the device in the object manager namespace, and a driver can either explicitly define a name or let the I/O manager generate one. Device objects are placed in the `[`\Device`]` directory in the namespace, which is inaccessible by user-space API. (Object manager is the executive component responsible for creating, deleting, protecting and tracking objects). If a driver needs to make it possible for applications to open the device object, it must create a symbolic link in the `[`\Global??`]` directory to the device object's name in the `[`\Device`]` directory.

Plug and Play drivers expose one or more interfaces by calling the `IoRegisterDeviceInterface` function, specifying a GUID that represents the type of functionality exposed. `IoRegisterDeviceInterface` determines the symbolic link that is associated with a device instance. An application wanting to open a device object represented with a GUID can call PnP setup functions in user-space, such as `SetupDiEnumDeviceInterfaces`, to enumerate the interfaces present for a particular GUID and to obtain the names of the symbolic links it can use to open

---

[2] Microsoft Driver Development Kit, Microsoft Corporation, 2006

the device objects. For each device reported by `SetupDiEnumDeviceInterfaces`, an application executes `SetupDiGetDeviceInterfaceDetail` to obtain additional information about the device, such as its name. After obtaining the device's name, the application can execute the Windows function `CreateFile` to open the device and obtain a handle.

4.3 Opening Devices

File objects are the kernel-mode constructs for handles to files or devices. When a caller opens a file or a device, the I/O manager returns a handle to a file object.

4.4 Driver Installation

The components involved in the driver's installation are shown in figure 4.2. Shaded objects in the figure correspond to components generally shipped with driver's installation files. The others are supplied by the system. If the PnP manager encounters a device for which no driver is installed, it relies on the user-mode PnP manager to guide the installation process. If the device is detected during the system boot, a devnode is defined for the device but the loading process is postponed until the user-mode PnP manager starts. The user-mode PnP manager is implemented in `[umpnpmgr.dll]` and runs as a service.

When a bus driver performs device enumeration, it reports device identifiers for the devices it detects back to the PnP manager. The identifiers are bus specific. For a USB bus, an identifier consists of a vendor ID and a product ID that the vendor assigned to the device. Together these IDs form what Plug and Play calls a device ID. The PnP manager also queries the bus driver for an instance ID to help it distinguish different instances of the same hardware. The instance ID can describe either a bus relative location or a globally unique descriptor. The device ID and instance ID are combined to form a device instance ID (DIID), which the PnP manager uses to locate the device's key in the enumeration branch of the Registry ($HKLM\backslash SYSTEM\backslash CurrentControlSet\backslash Enum$). The device's key contains subkeys Service and ClassGUID that help the PnP manager locate the device's drivers.

When device is connected to the bus, the bus driver informs the PnP manager about a device it enumerates using a DIID. The PnP manager checks the Registry for the presence of a corresponding function driver and when it does not find one, it informs the user-mode PnP manager about the new device by its DIID. The user-mode PnP manager first tries to perform an automatic installation without user

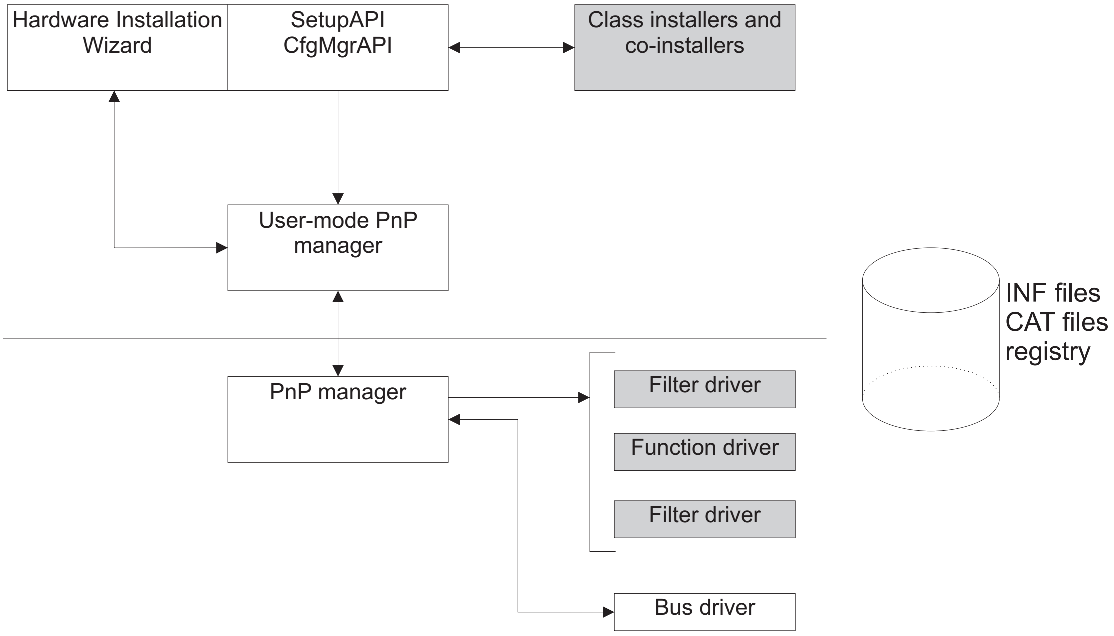

Figure 4.2: Components involved in the installation

intervention. If the installation process involves user interaction the user-mode PnP manager launches the `[rundll32.exe]` application to execute the Hardware Installation Wizard. The Hardware Installation Wizard uses SetupAPI and CfgMgrAPI to locate INF files that correspond to drivers that are compatible with the detected device. This process might involve having the user insert installation media containing a vendor's INF files or the wizard might locate a suitable INF file in the cabinet file that contains drivers that ship with Windows.

To find drivers for the new device, the installation process gets a list of hardware IDs and compatible IDs from the bus driver. These IDs describe all the various ways the hardware might be identified in a driver installation file. The lists are ordered so that the most specific description of the hardware is listed first. If some matches are found in multiple INFs, more precise matches are preferred over less precise matches.

The INF file locates the function driver's files and contains commands that fill in the driver's enumeration and class keys and the INF file might direct the Hardware Installation Wizard to launch class or device co-installer DLLs that perform class or device-specific installation steps.

4.4.1 Co-installers

A co-installer is a user-mode Win32 DLL. Typically, a co-installer performs installation tasks that require dynamic information that is not available when the INF for its device or device class is written.

4.5 Driver Loading and Initialization

The PnP manager begins device enumeration with a virtual bus driver called Root, which represents the entire computer system and acts as the bus driver for non-Plug and Play drivers and for the HAL. The HAL acts as a bus driver that enumerates devices directly attached to the motherboard as well as system components such as batteries. Instead of actually enumerating, the HAL relies on the hardware description the Setup process recorded in the Registry to detect the primary bus and devices such as batteries and fans. The primary bus driver enumerates the devices on its bus, possibly finding other buses, for which the PnP manager initializes drivers. Those drivers in turn can detect other devices, including other subsidiary buses. This recursive process of enumeration, driver loading and further enumeration proceeds until all the devices on the system have been detected and configured.

The PnP creates an internal tree called the device tree that represents the relationships between devices. Nodes in the tree are called devnodes and a devnode contains information about the device objects that represent the device.

A record of all the devices detected since the system was installed is recorded under the $HKLM\backslash SYSTEM\backslash CurrentControlSet\backslash Enum$ Registry key. Subkeys are in the form $\langle Enumerator\rangle \backslash \langle DeviceID\rangle \backslash \langle InstanceID\rangle$, where $\langle Enumerator\rangle$ is a bus driver, $\langle DeviceID\rangle$ is a unique identifier for a type of device and the $\langle InstanceID\rangle$ uniquely identifies different instances of the same hardware. [3]

---

[3] Russinovich, M. et al.: Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server, Microsoft Press, 2004, Chapter 9

Chapter 5

Linux Driver Model

5.1 Classes of Devices and Modules

The Linux way of looking at devices distinguishes between three fundamental device types: char devices, block devices and network devices. Each module usually implements one of these types.

Character Devices

A character device is one that can be accessed as a stream of bytes. A char driver is in charge of implementing this behavior. Char devices are accessed by filesystem nodes in the `[/dev]` directory.

Block Devices

A block device is a device that can host a filesystem. Linux allows the application to read and write a block device like a char device, it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel and thus in the kernel/driver interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user.

Network Interfaces

A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel. A network interface is not mapped to a node in the filesystem.

There are other ways classifying driver modules. In general, some types of drivers work with additional layers of kernel for a given type of a device. For example, one can talk of USB modules, serial modules and so on. Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device, a block device or a network device.

5.1.1 Loadable Modules

Linux kernel possesses ability to extend at runtime the set of features offered by the kernel. Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types of modules, including device drivers. Each module is made up of object code that can be dynamically linked to the running kernel by program `[insmod]` and can be unlinked by the program `[rmmod]`.

5.2 Device Model

The Linux device model is a complex data structure. The `kobject` is the fundamental structure that holds the device model together. The tasks handled by `struct kobject` and its supporting code are reference counting of objects, sysfs representation, hotplug event handling. It is rare for kernel code to create a standalone `kobject`. Instead kobjects are used to control access to larger objects, thus they are found embedded in other structures.

The `kobject` structure is often used to link together objects into a hierarchical structure that matches the structure of the subsystem being modeled. There are two separate mechanisms for this linking: parent pointer and ksets. The parent field in `struct kobject` is a pointer to another kobject - the one representing the next level up in the hierarchy. For example a `kobject` that represents a USB device, its parent pointer may indicate the object representing the hub into which the device is plugged. The main use for the parent pointer is to position the object in the sysfs hierarchy.

The main function of a kset is containment for kobjects. In fact, each `kset` contains its own `kobject` internally and it can be treated the same way as a `kobject`.

A subsystem is a representation for a high-level portion of the kernel as a whole. A subsystem, thus, is just a wrapper around a `kset` that contains a semaphore. This semaphore is used to serialize access to a kset's internal list. Every `kset` must belong to a subsystem.

Kobjects are the mechanism behind the sysfs virtual filesystem. For every directory in sysfs, there is a `kobject` living in the kernel. The sysfs filesystem has the usual tree structure, reflecting the hierarchical organization of the kobjects it represents.

5.2.1 Devices

At the lowest level, every device in a Linux system is represented by an instance of `struct device` structure. There are some interesting fields in this structure. The field `parent` holds a device to which this device is connected. In most cases a parent device is some sort of bus or host controller. The field `kobj` holds kobject that represents this device and links it into the hierarchy. The field `bus_id` is a string that uniquely identifies this device on the bus. The field `bus_type` identifies which kind of bus the device sits on. The field `device_driver` points to the driver that manages this device. The field `driver_data` may be used by the device driver to point to device's private data.

5.2.2 Device Drivers

The device model tracks all of the drivers known to the system. The main reason for this tracking is to enable the driver core to match up drivers with new devices. Once drivers are known objects within the system a number of other things become possible. Device drivers can export information and configuration variables that are independent of any specific device.

Drivers are defined by a `struct device_driver`. Once again, there are some interesting fields in this structure. Here, field `name` is the name of the driver, `kobj` is the inevitable kobject, `devices` is a list of all devices currently bound to this driver, `probe` is a function called to query the existence of a specific device, `remove` is called when the device is removed from the system and `shutdown` is called at the shutdown time of the driver.

Driver Structure Embedding

The `device_driver` structure is usually found embedded within a higher-level structure. To retrieve field `struct device_driver` from higher-level structure function `container_of` can be used.

5.3 Device Lifecycle

To better understand what the driver model does, let us walk through the steps of a device's lifecycle within the kernel. We describe how the USB subsystem interacts with the driver model, the basic concepts of how a driver is added and removed and how a device is added and removed from the system.

5.3.1 Add a Device

The USB subsystem declares a single `struct bus_type` called `usb_bus_type`. This `usb_bus_type` variable is registered with the driver core when the USB subsystem is loaded in the kernel with a call to `bus_register`. When that happens, the driver core creates a sysfs directory in `[/sys/bus/usb]` that consists of two directories: devices and drivers. All USB drivers must define a `struct usb_driver` variable that defines the different functions that this USB driver can do.

That structure contains a `struct device_driver` that is then initialized by the USB core when the USB driver is registered with driver core.

The USB core, with help from the architecture-specific code that actually talks to the USB bus, starts looking for all USB devices. When a USB device is found, the USB core creates a new variable in memory of type `struct usb_device`.

The bus-specific fields of this USB device are initialized by the USB core, and the `struct device` parent variable is set to the USB bus device that this USB device lives on. After the USB device structure is initialized, the device is registered with the driver core with a call to `device_register`.

Within the `device_register` function, the driver core initializes a number of device's fields, registers the device's `kobject` with the kobject core (this causes a hotplug event), and then adds the device to the list of devices that are held by the device's parent.

The device is then added to the bus-specific list of all devices. Then the list of all drivers that are registered with the bus is walked and the `match` function of the bus is called for every driver, specifying this device.

The `match` function looks at the USB device-specific information of the device and driver to see if the driver states that it can support this kind of device. If the match is not successful, the function returns 0 back to the driver core and the driver core moves on to the next driver in its list.

If the match is successful, the function returns 1 back to the driver core. This causes the driver core to set the driver pointer in the `struct device` to point to this driver and then it calls the `probe` function that is specified in the `struct device_driver`.

If the USB driver's `probe` function determines that it can not handle this device for some reason, it returns a negative error value, which is propagated back to the driver core and causes it to continue looking through the list of drivers to match one up with this device. If the `probe` function can claim the device, it does all the initialization that it needs to do to handle the device properly and then it returns 0 back up to the driver core. This causes the driver core to add the device to the list of all devices currently bound by this specific driver and creates a symlink within the driver's directory in sysfs to the device that it is now controlling. This symlink allows users to see exactly which devices are bound to which drivers.

5.3.2 Remove a Device

A USB device can be removed from a system at any time. When a USB device is to be removed, function `device_unregister` is called with a pointer to the `struct usb_device`'s struct device member.

In the `device_unregister` function, the driver core unlinks the sysfs file from the driver bound to the device, removes the device from its internal list of devices, and calls `kobject_del` with a pointer to the struct kobject that is contained in the `struct device` structure. That function makes a hotplug call to user space stating that the `kobject` is now removed from the system and it deletes all sysfs files associated with the `kobject` and the sysfs directory itself that the `kobject` had originally created.

The `kobject_del` function also removes the kobject reference of the device itself. If that reference was the last one, then the `release` function for the USB device itself is called. That function frees up the memory that the `struct usb_device` took up.

5.3.3 Add a Driver

A USB driver is added to the USB core when it calls the `usb_register_driver` function. This function initializes the `struct device_driver` structure that is contained within the `struct usb_driver` structure. Then the USB core calls the `driver_register` function in the driver core with a pointer to the struct `device_driver` structure contained in the `struct usb_driver` structure.

The `driver_register` function initializes a few locks in the `struct device_driver` structure and then calls the `bus_add_driver` function. This function looks up the bus that the driver is to be associated with, the driver's sysfs directory is created. The bus's internal lock is grabbed and then all devices that have been registered with the bus are walked and the match function is called for them, just like when a new device is added. If the match function succeeds, then the rest of the binding process occurs, as described in the previous section.

5.3.4 Remove a Driver

Removing a driver is a very simple action. For a USB driver, the driver calls the `usb_unregister_driver` function. This function calls the driver core function `driver_unregister`, with a pointer to the `struct device_driver` portion of the `struct usb_driver` structure passed to it. The `driver_unregister` function handles cleaning up sysfs attributes that were attached to the driver's entry in the sysfs tree. It then iterates over all device that were attached to this driver and calls the release function for it. Then it waits for all reference counts on this driver to be dropped to 0 before it is safe to return. This is needed because the `driver_unregister` function is most commonly called as the exit path of a module that is being unloaded. The module needs to remain in memory for as long as the driver is being referenced by devices and by waiting for this lock to be freed, this allows the kernel to know when it is safe to remove the driver from memory.

5.4 Hotplug

There are two different ways to view hotplugging. The kernel views hotplugging as an interaction between the hardware, the kernel and the kernel driver. Users view hotplugging as the interaction between the kernel and user space through the program called hotplug. This program is called by the kernel when it wants to notify user space that some type of hotplug event has just happened within the kernel. [4]

This program is typically a very small bash script that passes execution on to a list of other programs that are place in the `[/etc/hotplug.d/]` directory tree.

As mentioned previously `[/sbin/hotplug]` is called whenever a kobject is created or destroyed. The hotplug program is called with a single command-line argument providing a name for the event. The kernel and specific subsystem involved also set a series of environment variables with information what has just occurred. Default environment variables set by the kernel are:

- ACTION - string add or remove, depending on whether the object was just created or destroyed.
- DEVPATH - a directory path, within the sysfs fylesystem.
- SEQNUM - sequence number for this hotplug event.
- SUBSYSTEM - subsystem name involved, for USB devices it contains string usb.

---

[4] Corbet, J. et al.: Linux Device Drivers Third Edition, O'Reilly, 2005, p. 397

The USB subsystem always adds the following environment variables

- PRODUCT - a string in the format idVendor/ idProduct/bcdDevice
- TYPE - a string in the format bDeviceClass/bDeviceSubClass/bDeviceProtocol
- INTERFACE - a string in the format bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol. This variable is set if bDeviceClass field is set to 0.

Chapter 6

Wine

6.1 Wine Architecture

Wine is an implementation of the Windows API and can be used as a library to port Windows applications to UNIX systems. Wine's main task is to run Windows executables under non Windows operating systems. Wine implementation is closer to the Windows NT architecture as illustrated in figure 6.1.

Wine completely replaces `[kernel32.dll]`, `[user32.dll]` and `[gdi32.dll]` with its own implementation. All other DLLs on top of these three DLLs fully depend on them. Since Wine is leaning towards the NT way of implementing things, the NTDLL is another core DLL that is implemented in Wine and much of KERNEL32 and ADVAPI32 functionality is implemented through the NTDLL.

Role of Wine server in the whole design is to provide the backbone for the implementation of the core DLLs. It mainly implements inter-process communication, synchronization, process/thread management and object sharing. It can be seen, from functional point of view as a NT kernel.

When the Wine server launches, it creates a UNIX socket for the current host. All Wine processes launched later will connect to the Wine server using this socket. If a Wine server was not already running, the first Wine process will start up the Wine server in auto-terminate mode (once the last client is disconnected it will terminate itself). Every thread in each Wine process has its own request buffer, which is shared with the Wine server. When a thread needs to synchronize or communicate with any other thread or process, it fills out its request buffer, and then writes a command code through the socket. The Wine server handles the command as appropriate, while the client thread waits for a reply. The Wine server itself is a single separate UNIX process and does not have its own threading - instead it is a simple poll loop that alerts the Wine server whenever anything happens. Because the Wine

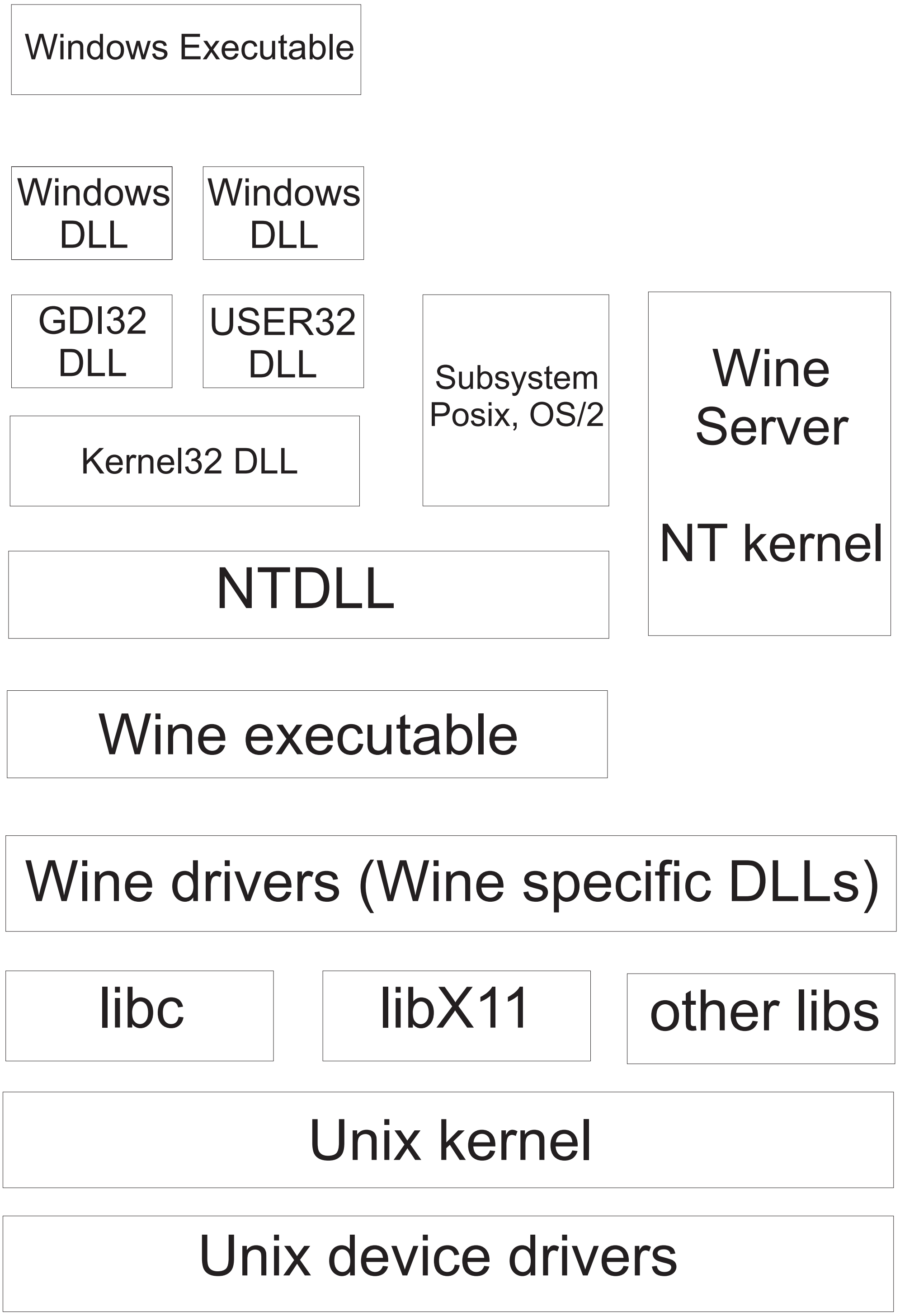

Figure 6.1: The WINE Architecture

server needs to manage processes, threads, shared handles, synchronization and any related issues, all the clients' Win32 objects are also managed by the Wine server, and the clients must send requests to the Wine server whenever they need to know any Win32 object handle that is associated with UNIX file descriptor.

Wine uses the UNIX drivers to access the various hardware on the box. However, in some cases, Wine provides a driver in Windows sense to a physical hardware device. This driver is in fact a proxy to the UNIX driver. [5]

6.1.1 Wine Dlls

Each DLL is implemented in a UNIX shared library. The file name of this shared library is the module name of the DLL with a .dll.so suffix. This shared library contains the code itself for the DLL, as well as some information, as the DLL resources and a Wine specific DLL descriptor. The descriptor, when DLL is instantiated, is used to create an in-memory PE header. Details about DLLs file can be found in chapter 2.

The DLL descriptor and entry point table is generated by the `[winebuild]` tool, taking DLL specification files with the extension .spec as input. Resource or message tables are also added to the descriptor by `[winebuild]`.

If the application wants to import a DLL, Wine will search for the DLL in the following order till it finds it:

- list of registered DLLs (loaded both native libraries and shared libraries with its DLL descriptors)
- Wine will look for it on the disk. First it will try to find shared library and if it fails, it will look for a native Windows DLL.

After the DLL has been identified, it has been mapped into memory using a `dlopen` call. Wine does not use the shared library mechanisms for resolving and/or importing functions between two shared libraries. The shared library is only used for providing a way to load a piece of code on demand. This piece of code, thanks the DLL descriptor will provide the same type of information a native DLL would. Wine is using the same code for native and builtin DLL to handle imports and exports.

Wine also relies on the dynamic loading features of the UNIX shared libraries to relocate the DLL if needed. Since Wine is 32bit code itself and if the compiler supports Windows' calling convention (stdcall), Wine can resolve imports into Win32

---

[5] Wine Developer's Guide, Wine Community, 2006, p. 66

code by substituting the addresses of the wine handlers directly without any layer in between.

6.1.2 File Management

Windows API implementation comes closer to the UNIX paradigm "Everything is a file". Access to devices in Windows is done through the same API as access to ordinary files.

In Wine implementation of Windows interface there is need to map a file name into a file name in the UNIX world. In the following lines we will only focus on device names mapping. Access rights to all files and regular file names mapping can be found in [2].

Devices are manipulated in Windows with both read and write operations, but also control mechanisms. Since, this is also supported in Linux, there is also a need to open a device when given a Windows device name.

Every device path is of the following form `[/??/devicename]`. As Windows device names are case insensitive, Wine also converts them to lower case before any operation. Then, the first operation Wine tries is to check whether `[$(WINEPREFIX)/dosdevices/devicename]` exists. If so, it is used as the final UNIX path for the device. The configuration process is in charge of creating symbolic links pointing to real device file (for example /dosdevices/physicaldrive0 pointing to /dev/hda0). If such a link cannot be found and the device name looks like a DOS disk name (like c:), Wine first tries to get the Unix device from path `[$(WINEPREFIX)/dosdevices/c:]` (i.e. the device which is mounted on the target of the symbolic link). If this does not give a UNIX device, Wine tries whether `[$(WINEPREFIX)/dosdevices/c:]` exists. If so, it is assumed to be a link to the actual UNIX device. If it does not exist, Wine tries to get the UNIX device from the system information. If the devicename is `NULL`, then `[/dev/null]` is returned. If the devicename is a default serial name (COM1 up to COM9) or default parallel name (LPT1 up to LPT9), then Wine tries to open the Nth serial or printer in the system. Some basic old DOS name support is done and the whole process is retried with those new names. Summary of described mappings is in table 6.1.

Operations on Files

Reading and Writing

Wine server is involved in any read or write operation, as Wine needs to transform the Windows handle to the file into a UNIX file descriptor it can pass to any UNIX file function. But the reading and writing is done on client side by calls to

Table 6.1: Mapping of device names

Windows device name	NT device name	Mapping to Unix device name
⟨*any_path*⟩ *NUL*	\Global??\NUL	/dev/null
\\.\E:	\Golbal??\E:	$(WINEPREFIX)/ dosdevices/e::, if link exists, guess the device from the system, if link does not exist
\\.\ ⟨*device_name*⟩	\Global??\ ⟨*device_name*⟩	$(WINEPREFIX)/ dosdevices/ ⟨*device_name*⟩, if the link exists

UNIX equivalents of functions for reading and writing. The user, from the UNIX perspective, running the Wine executable must have read or write access respectively to the device.

Locking

Windows provides file locking capabilities. When a lock is set it controls how other processes in the system will have access the range in the file. The implementation of locking is done in wineserver.

I/O Control

Wine has not implemented support for files and directories for IO control. But it implements IO control for specific devices such as disks and cdroms.

Buffering

Wine does not do any buffering, but rely on the underlying UNIX kernel for that. Doing lots of small reads on the same file can turn into a performance hit, because each read operation needs a round trip to the wineserver in order to get a file descriptor.

Chapter 7

Microsoft STI

The imaging architecture in Windows 2000 and Windows 95/98 consists of a low-level hardware abstraction - STI and a high-level set of APIs known as TWAIN. In Windows XP and Windows ME Microsoft introduced Windows Imaging Architecture (WIA), which is built on top of STI. This document will describe TWAIN/STI stack which was also implemented in Linux environment.

STI architecture also supports the so called "push model" of events handling. New types of scanners can notify imaging subsystem about events generated in device (e.g. button press). This event is propagated through push model into system. These events are handled in STI Event Monitor. Push model feature will not be supported in Linux implementation of STI.

7.1 Core Components

Main component of the Still Image imaging architecture is a dynamic library called `[sti.dll]`. There are four types of components that communicate with `[sti.dll]` library, as shown in picture 7.1.

One of it is a STI class installer (`[sti_ci.dll]`). Functionality in Still Image devices class installer is invoked only when a new Still Image device is installed or removed.

A Still Image Event Monitor (`[stimon.exe]`/`[stisvc.exe]`) monitors all installed Still Image devices and receives notification when STI device event occurs. An event typically indicates that a device is ready to transmit image data. The Event Monitor also keeps track of all registered applications and can start an application when an event is detected.

Scanners and Cameras Control Panel application is represented by `[sticpl.cpl]` component. This component is invoked only for configuration pur-

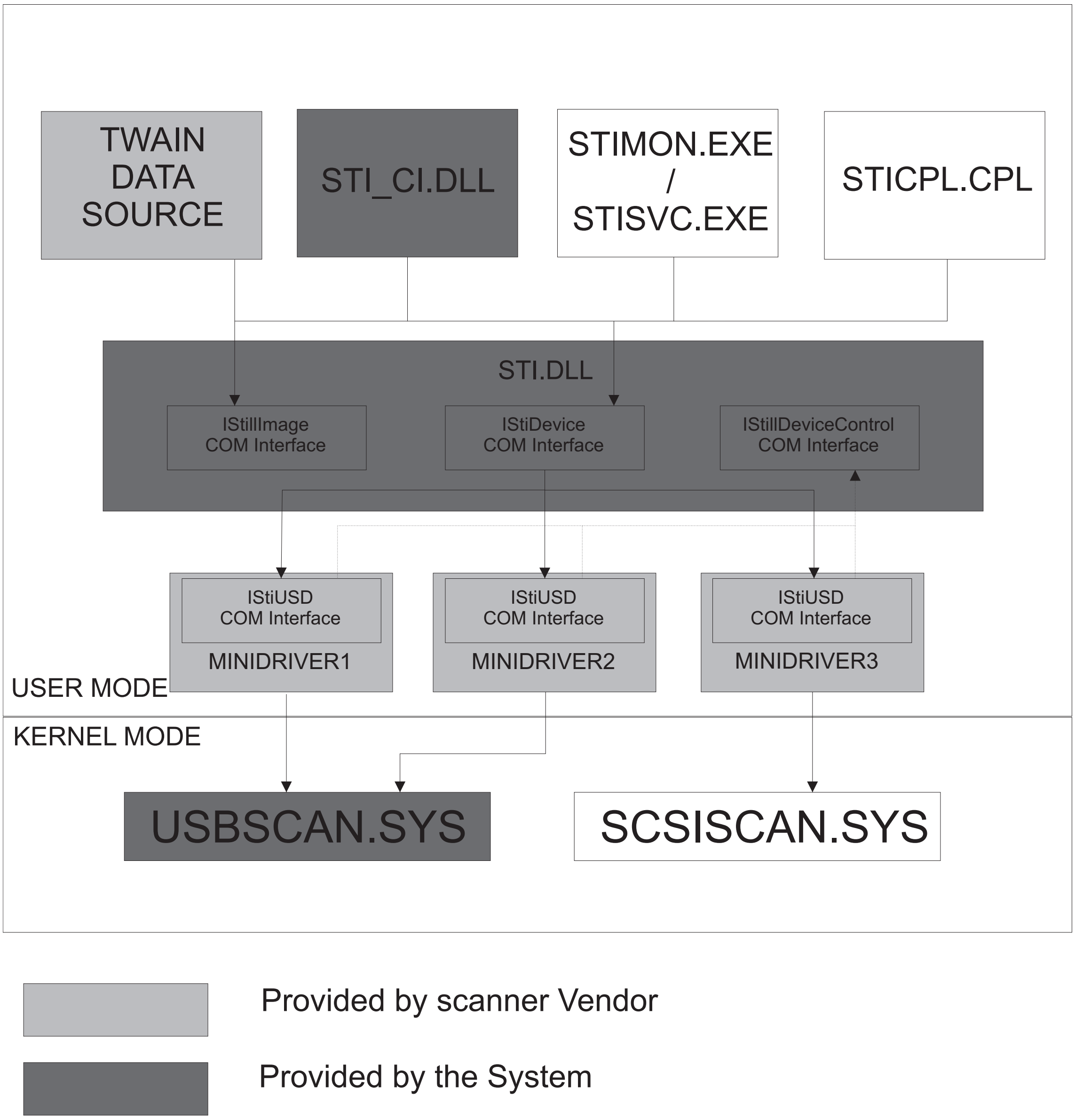

Figure 7.1: The STI architecture

poses from Windows control panel. It allows users to assign specific Still Image device events to specific applications. In this way, the Event Monitor will know which application to start when it detects an event. It also lets users test STI devices.

The TWAIN DataSource is a component of the TWAIN Scanning architecture described in chapter 8.

STI dynamic library communicates with appropriate user mode Still Image minidrivers. These minidrivers can detect device activity and notify the Still Image Event Monitor of that activity through Still Image device events. These minidrivers also pass image data from kernel-mode drivers to upper level software.

User-mode Still Image minidrivers are vendor supplied components that provide a device-specific, user-mode interface to an appropriate kernel-mode driver. Each of these user-mode drivers must implement the `IStiUSD` COM interface. They communicate with kernel-mode drivers by calling the `CreateFile`, `ReadFile`, `WriteFile` and `DeviceIoControl` Win32 functions.

7.1.1 STI COM Interfaces

Microsoft STI defines several Still Image COM interfaces that allow STI components to communicate with each other.

IStillImage COM Interface

The `IStillImage` COM interface provides access to Still Image Event Monitor. Applications can register themselves as push-model aware or obtain information about the system's Still Image devices. The interface provides some application management functions, such as enabling event notification and starting an application for use by customized application control software. Additionally, the `IStillImage` interface provides access to the `IStiDevice` COM interface, which allows applications to perform I/O operations on Still Image devices.

IStiDevice COM Interface

The `IStiDevice` COM interface provides applications with the ability to communicate with Still Image devices. Interface methods allow applications to send and receive data and commands, to run diagnostic tests, to receive notifications of Still Image device events and to obtain device capabilities and status information.

Access to the `IStiDevice` interface is obtained by calling the `CreateDevice` method of the `IStillImage` COM interface. Many of the `IStiDevice` interface's methods are implemented by calling like-named methods defined by the `IStiUSD` COM interface.

IStiUSD COM Interface

The `IStiUSD` COM interface is the means by which the `IStiDevice` COM interface communicates with Still Image devices. The `IStiUSD` interface's methods are implemented by each vendor-supplied user-mode Still Image minidriver. Still Image minidrivers typically implement `IStiUSD` interface methods by calling the appropriate kernel-mode driver. Each minidriver must define all interface methods, but if a method is not needed it can return unsupported status.

IStiDeviceControl COM Interface

The `IStiDeviceControl` COM interface provides user-mode Still Image minidrivers with access to information stored within the Still Image Event Monitor. It also allows minidrivers to write information into the Still Image error log.

7.2 Installing and Configuring Still Image Components

7.2.1 Still Image Devices Class Installer

Microsoft provides a default class installer for Still Image devices with support for special INF file entries enumerated in table 7.1. The default class installer supports vendor-supplied co-installer extensions. If necessary vendors can provide customized installation programs that can be used instead of the Microsoft-supplied class installer. The default class installer for Still Image devices, `[sti_ci.dll]`, recognizes a special set of INF file entries. Within an INF file, these entries must be placed within a device's INF DDInstall section.

The default INF file for USB Still Image devices, defines two installation sections for each device type:

1. Needs=STI.USBSection
2. Needs=STI.USBSection.Services

These sections are defined in file STI.INF. Clause Include=sti.inf is therefore required in vendor supplied INF file.

Table 7.1: STI INF sections

INF file entry	Value	Comments
Subclass	StillImage	Required
DeviceType	1 for scanners, 2 for cameras	Required
DeviceSubType	Vendor-defined value	Optional
Connection	For non-PnP devices	Optional
Capabilities	Flags identifying device capabilities	Optional
PropertyPages	identifies the name and entry point of a DLL that creates customized property sheet pages for STI devices	Optional
DeviceData	Identifies a vendor-supplied data section containing information to be stored in the Registry, under the DeviceData key. For TWAIN-supported devices, this key must contain a TwainDS entry	Optional
Events	Identifies a vendor-supplied data section listing Still Image device events	Optional
UninstallSection	Points to an INF section typically containing INF delfiles directives and INF delreg directives	Optional

7.2.2 Registry Entries for Still Image Devices

Non-modifiable Registry Entries

Microsoft STI defines some registry entries that are used internally by Still Image Subsystem. The table 7.2 lists all of the types of registry entries that should not be modified by vendor software.

Table 7.2: STI registry entries

Registry key and Definition
HKLM\SYSTEM\CurrentControlSet\Control\StillImage\Logging\STICLI
Specifies which vendor-generated messages are written to the Still Image log file. Can be any combination of the following bit masks: 0x1 info messages, 0x2 warn messages, 0x4 error messages
HKLM\SYSTEM\CurrentControlSet\Control\StillImage\Logging\STIMON
Specifies which Event Monitor messages are written to the Still Image log file. Can be any combination of the following bit masks: 0x1 info messages, 0x2 warn messages, 0x4 error messages
HKLM\SYSTEM\CurrentControlSet\Control\Class\{6bdd1fc6-810f-11d0-bec7-08002be2092f}
Contains information about installed Still Image devices
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\StillImage\RegisteredApplications
Contains a list of registered imaging applications
HKLM\SYSTEM\CurrentControlSet\Control\DeviceClasses\{6bdd1fc6-810f-11d0-bec7-08002be2092f}
Contains information about installed Still Image device interfaces

Vendor-modifiable Registry Values

Still Image architecture defines several registry entries. Some of them can be modified by vendor-supplied components. They are defined in `[stireg.h]`.

Chapter 8

TWAIN

TWAIN defines a standard software protocol and API for communication between software applications and image acquisition devices.

8.1 The TWAIN Architecture

The TWAIN architecture consists of four layers (Application, Protocol, Acquisition, Device). These layers are occupied by TWAIN software elements (application, Source Manager, Source) as shown in figure 8.1.

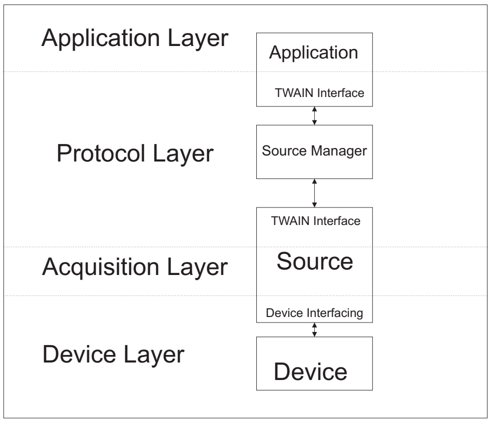

Figure 8.1: TWAIN layers

8.1.1 Application Layer

TWAIN describes user interface guidelines for the application developer regarding how users access TWAIN functionality and how a particular Source is selected. TWAIN is not concerned about how the application is implemented.

8.1.2 Protocol

The protocol is the language spoken and syntax used by TWAIN. It implements precise instructions and communications required for the transfer of data. This layer includes:

- Part of application software that provides the interface between the application and TWAIN
- The TWAIN Source Manager provided by TWAIN
- Part of Source that provide interface to Source Manager to receive instructions and transfer back data and Return Codes.

8.1.3 Acquisition

Acquisition devices may be physical (e.g. scanner) or logical (e.g. image generator). The software elements written to control acquisitions are called Sources and reside primarily in this layer. The Source transfers data for the application. It uses the format and transfer mechanism agreed upon by the Source and application.

The Source always provides a built-in user interface that controls the device the Source was written to drive. An application can override this and present its own user interface for acquisition.

8.1.4 Device

TWAIN is not concerned with the device layer at all. The Source hides the device layer from the application. The Source provides the translation from TWAIN operations and interactions with the Source's user interface into the equivalent commands specific for the device driver that cause the device to behave as desired.

8.1.5 Communication between TWAIN Elements

Communication between TWAIN elements is possible through two entry points. They are called `DSM_Entry()` and `DS_Entry()`.

The Application

The goal of the application is to acquire data from a Source. However, applications cannot contact the Source directly. All requests for data must be handled through the Source Manager.

TWAIN defines operations through operation triplets that are passed to `DSM_Entry()` function. The application specifies in each operation triplet which element, Source Manager or Source, is the final destination for requested operation. Function `DSM_Entry()` is the only TWAIN entry point for application.

The parameter list of the `DSM_Entry()` function contains:

- An identifier structure providing information about the application that originated the function call
- The destination of this request (Source Manager or Source)
- A triplet that describes the requested operation.
- A pointer field to allow the transfer of data

Operation Triplets

- Data group for the operation (`DG_`)
- Data argument type for the operation (`DAT_`)
- Message for the operation (`MSG_`)

The desired action is defined by an operation triplet passed as three parameters in the function call. Each triplet uniquely and without ambiguity specifies a particular action. No operation is specified by more than a single triplet. The three parameters that make up the triplet are Data group, Data argument type and Message ID. Each parameter conveys specific information.

Data Group (DG_XXX)

Operations are divided into groups identified by data group identifier. There are currently two defined in TWAIN:

- DG_CONTROL - these operations involve control of the TWAIN session.
- DG_IMAGE - these operations work with image data.

Data Argument Type (DAT_XXX)

This parameter of the triplet identifies the type of data that is being passed or operated upon. The argument type may refer to a data structure or a variable the `pData` pointer argument is pointing at.

Message ID (MSG_XXX)

This parameter identifies the action that the application or Source Manager has to take.

The complete list of all TWAIN operation triplets can be found in [3].

The Source Manager

The Source Manager provides the communication path between the application and the Source. It supports the user's selection of a Source and loads the Source for access by the application. Communication from application to Source Manager arrive in the `DSM_Entry()` entry point.

- If the destination in the `DSM_Entry` call is the Source Manager, then the Source Manager processes the operation itself.
- If the destination in the `DSM_Entry` call is the Source - The Source Manager translates the parameter list of information, removes destination parameter and calls appropriate Source by calling its `DS_Entry()` function. TWAIN requires each Source to have this entry point.

The Source Manager can initiate three operations that were not originated by the application. These operation triplets exist just for communications between Source Manager and Source and are executed by the Source Manager while it is displaying its Select Source dialog box. These operations are used to identify the available Sources and to open or close Sources.

The Source

The Source receives operations either from the application, via the Source Manager or directly from the Source Manager. It processes the request and returns the appropriate return code indicating the results of the operation to the Source Manager. If the originator of the operation was the application then the return code is passed back to the application as the return value of its `DSM_Entry()` function call. If the operation was unsuccessful a condition code containing more specific information is set by the Source. Although the condition code is set, it is not automatically passed back. The application must invoke an operation to inquire about the contents of the condition code.

Communication Flow from Source to Application

The majority of operation requests are initiated by the application and flow to the Source Manager and Source. The Source, via the Source Manager, is able to pass data back and return codes. However there are four times when the Source needs to interrupt the application and requests that an action occurred. These notices are presented to the application in its event loop.

- Notify the application that a data transfer is ready to occur
- Request that the Source's user interface is closing
- Notify the application that the OK button bas been pressed, accepting the changes the user has made
- A device event has occurred

8.2 The User Interface

When an application uses TWAIN to acquire data, the acquisition process may use following user interfaces:

- Application user interface - Allow user select the device from which to acquire the data.
- Source Manager user interface - Provides list of available Sources - devices. User can choose one of the devices to acquire data from it. If desired, the application can write its own version of this user interface.
- Source user interface - Every TWAIN Source provides a user interface specific to its particular device. If desired, the application can write its own version of this user interface.

8.3 The TWAIN State Protocol

The application, Source Manager and Source must communicate to manage the acquisition of data. The process occurs in a particular sequence as illustrated in figure 8.2. The TWAIN protocol defines seven states to ensure the sequence is executed correctly. These states exist in a TWAIN session. First session is defined as a period while an application is connected to a particular Source via the Source Manager. The second unique session is the period while an application is connected to the Source Manager. At a given point in a session the TWAIN Source and the Source Manager occupy a particular state. Transitions to a new state are caused by operations requested by the application or Source. State transitions can be in a forward or backward direction. Most of the transitions are single-state transitions, but there are situations where two-state transition may occur.

The states can be divided into two groups:

- States that are occupied only by Source Manager - that are states 1, 2 and 3
- States that are occupied only by Sources - that are states 4, 5, 6 and 7

The Source Manager and Sources can not occupy other states than the enumerated above. If application uses multiple Sources then each connection is a separate session and each Source resides in its own state without regard for what state the other Sources are in.

8.3.1 Description of States

State 1 - Pre-Session

The Source Manager resides in this state before application establishes a session with it. At this point the Source Manager is not loaded into memory.

State 2 - Source Manager Loaded

The Source Manager is now loaded into memory. It is not open yet. At this time the Source Manager is prepared to accept operations from the application.

State 3 - Source Manager Open

The Source Manager is open and ready to manage Sources. The Source Manager is now prepared to provide lists of Sources, to open Sources and to close Sources. The Source Manager will remain in State 3 for the remainder of the session until it is closed. It can not be closed while the application has any Sources opened.

State 4 - Source Open

The Source has been loaded and opened by Source Manager in response to an operation from the application. It is ready to receive operations. The Source should have verified now that sufficient resources exist for it. The application can inquire about the Source's capabilities and also to set those capabilities to its desired values. Inquiry of capabilities can occur while Source is in states 4, 5, 6 and 7, but setting values to capabilities can occur only in state 4.

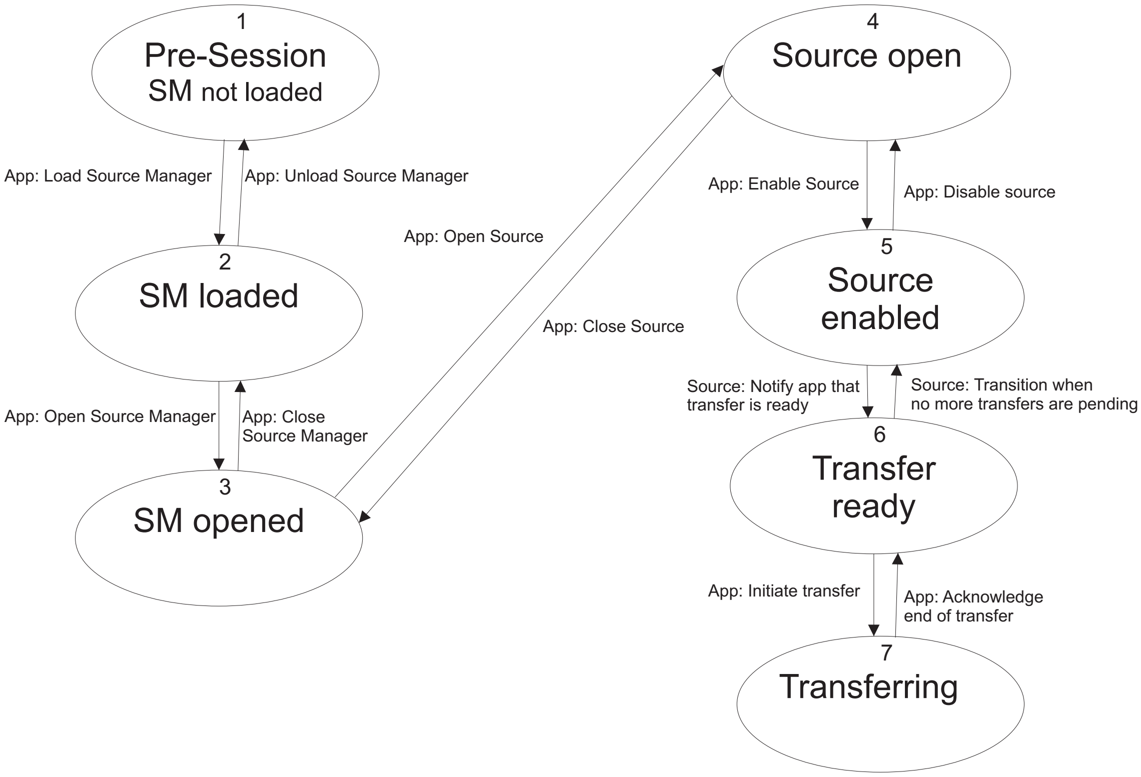

Figure 8.2: TWAIN state transitions

State 5 - Source enabled

The Source has been enabled by an operation from application via the Source Manager and is ready for transfers. If the application has allowed the Source to display its user interface, the Source will do that when it enters this state.

State 6 - Transfer is Ready

The Source is ready to transfer one or more data items to the application. The transition from state 5 to 6 is triggered by the Source notifying the application that the transfer is ready. Before initiating the transfer the application must inquire information about the image.

State 7 - Transferring

The Source is transferring the image to the application. The transfer mechanism being used was negotiated during state 4. The transfer will either complete successfully or terminate. The Source sends appropriate return code indicating result of the transfer.

8.4 Capabilities

Developers of applications need to be aware of a Source's capabilities and may influence the capabilities that the Source offers to the application's users. To do this, the application can perform capability negotiation:

- Determine - if the selected Source supports a particular capability.
- Inquire - about the current value for particular capability, its default value and set of available values that are supported by the Source.
- Request - that the Source set concrete value to the application's desired value.
- Limit - if needed, the Source's available values to a subset of what would normally be offered.
- Verify - that the new values have been accepted by the Source.

8.4.1 Available Modes for Data Transfer

There are three different modes that can be used to transfer data from the Source to the application: Native, Disk file and Buffered memory.

Native

Every Source must support this transfer mode. It is the default mode and is the easiest for an application to implement, but the format of the data is platform-specific. The Source allocates a single block of memory and writes the image data into the block. It passes a pointer of the memory location to the application. The application is responsible for freeing the memory after the transfer.

Disk File

A Source is not required to support this transfer mode but it is recommended. The application creates the file to be used in the transfer and ensures that it is accessible by the Source. File format can be negotiated through capabilities negotiation.

Buffered Memory

Every Source must support this transfer mode. The transfer occurs through memory using one or more buffers. Memory for the buffers are allocated and deallocated by the application. The data are transferred as an unformatted bitmap. The application must use information available during the transfer to learn about each individual buffer and be able to correctly interpret the bitmap. [6]

---

[6] TWAIN Specification, TWAIN Working Group Committee, 2000, p. 22

Chapter 9

Implemented Code

Goal was to use existing Windows drivers and create an environment around them that will resemble original Microsoft Windows environment to let the driver drive the device. We focused on USB scanner devices. Following are identified requirements for scanner drivers to work in Linux environment:

- Implementation of Microsoft Windows Still Image Architecture in Linux environment that will provide the same interface to scanner device drivers as Windows would.
- Implementation of interface that will provide Still Image implementation access to USB scanners.
- Implementation of high-level API for scanning subsystem.

Figure 9.1 illustrates all components involved in Microsoft's Windows scanning architecture and the interactions between these components. All user-space components depend on Win32 API. Windows application component stands for any application that provides acquisition functionality from scanner devices and uses scanning subsystem.

Image acquisition subsystem component covers all system scanning functionality, including low level API (Still Image) and high-level API (TWAIN or WIA).

Device driver component covers all driver libraries and/or kernel system libraries that are shipped with driver for specific device.

Kernel driver component stands for all kernel drivers in the system that are fundamental in communication of device driver with scanner device.

Device component stands for scanner device itself.

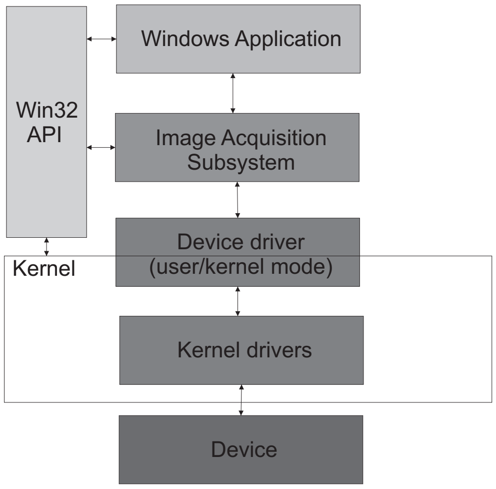

Figure 9.1: Overview of scanning components

9.1 Implementation Decisions

9.1.1 USB Scanner Driver

The skeleton for the USB scanner driver was taken from Linux kernel source tree. It was updated with specific code to simulate behavior of the original Microsoft's `[USBSCAN.SYS]` driver.

The USB scanner driver communicates with scanners through control and bulk endpoints. Support for communication through control endpoints were added to the skeleton. Communication through bulk endpoints was already implemented in the skeleton.

Windows USB drivers access devices through files created in device tree. For specific device there can be found device files for device and for every device's endpoint in the device tree. Windows drivers access the endpoints through these device's endpoint files. Our USB scanner implementation is creating a device file for each endpoint in `[/dev]` directory that points to original device file that was created by the kernel in the `[/dev]` directory. The requests from Windows' scanner drivers are properly mapped to the device files. Note that Wine supports symlinks from `[$(WINEPREFIX)/dosdevices]` directory to `[/dev]` directory, that is described in 6. However, we chose not to use this mechanism, because it requires another manual step in scanner configuration.

Support for handling USB ioctl codes that originate in Windows scanner drivers was added to simulate properly USB device communication. Adjustment can be found in function `ioctl()` in module `[usbscan.c]`.

The driver has one functionality restriction that comes from architecture. There is no way how our USB scanner driver can communicate with Wine or query the Wine's Registry. When a new scanner is installed then the Linux usbscan driver will create files for the scanner in `[/dev]` directory. These files will be of form: `[USBSCAN0]`, [$USBSCAN0\backslash 0$], [$USBSCAN0\backslash 1$], [$USBSCAN0\backslash 2$]. But there is no way how to check Wine's Registry for user that will try to scan from Wine, thus we are not sure that the scanner installed in Linux as scanner 0 will be installed in Wine also as scanner 0.
Note that every user that runs Wine uses different Registry (Registry loaded from different files).

9.1.2 Wine as Win32 API Wrapper

Wine is an open source implementation of Win32 API on top of Unix system. The decision was to extend this implementation to provide environment for scanner device drivers. Identified key components from Windows are:

- GUI - the whole graphical user interface. TWAIN DataSources do not separate driver access from graphical representation.
- SETUPAPI - scanner driver installation uses SETUPAPI during its installation phase. Though there is still missing implementation for a lot of SETUPAPI functions in Wine. Our goal was not to implement the whole SETUPAPI, however we implemented some of them to satisfy our requirements.
- Registry - Possibly every Windows piece of functionality depends on Registry and Microsoft's Registry are vastly used by drivers.
- COM interfaces - Still Image architecture is based on COM interfaces.
- Windows kernel - Wine server implements part of Windows kernel that is used by Win32 API.

At first look it seems that these identified components can be extracted from Wine and used separately to create environment around Windows scanner drivers. Identified components can depend on other components, finally it ends up with extracting considerable part of Wine.

Also it seems that there could be a possibility that the installation information contained in INF files can be parsed and used to create Registry entries for scanner.

Similar approach was used by NDIS wrapper to load particular DLL for network card driver. But Windows scanner drivers are often set of DLLs that can be loaded also with the help of the information stored in the Registry, thus the installation process could be hard without SETUPAPI. The use of other components during normal driver installation can make INF parsing hard, even impossible.

9.1.3 Scanning Subsystem Implementation

Microsoft Windows' scanning subsystem concept is shown in figure 7.1. Core of the scanning subsystem is represented by Still Image component. This is a core component that is employed with every scanner driver even with other scanning components. We have implemented this component in Linux environment. Details about its implementation decisions are described in 9.1.6

On the top of STI resides WIA and/or TWAIN component. WIA is new type of scanning architecture introduced by Microsoft in Windows XP. It is a complex subsystem with its own Windows service.

This new concept of scanning is not spread between scanner vendors yet, thus only minimum of all scanner drivers works with WIA. Even if there exists WIA scanner driver there exists also TWAIN driver for the same device.

TWAIN is designed as an open standard, simple to use. Every Windows scanner driver that conforms to TWAIN standard uses more or less Microsoft's STI functionality. Wine provides implementation for TWAIN and also original TWAIN DLls can be used. Wine's implementation of TWAIN 32bit version of DLL will load only one DataSource provided by Wine. To allow load another TWAIN Data Source, code need to be changed and recompiled. Place, where changes are needed, is in file [$\$(WINE\_SOURCE\_DIR)\backslash dlls\backslash twain\_32\backslash dsm\_ctrl.c$] in function `twain_autodetect()`. To add new driver following line needs to be added: $twain\_add\_onedriver(''c:\backslash windows\backslash system32\backslash sti-twain.ds'')$.
And lines:

```
twain_add_onedriver("gphoto2.ds");
twain_add_onedriver("sane.ds");
```

need to be commented out.

Note that now it will suffice to create only symlink that will be called `[sti-twain.ds]` that will point to installed Data Source.

For testing purposes Wine's implementation of TWAIN was used, with patched DataSources that will load.

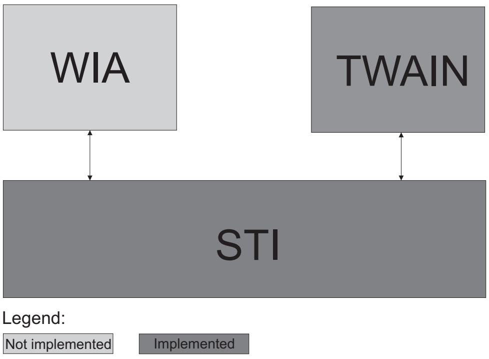

Figure 9.2: TWAIN and WIA

9.1.4 STI Interface to USB Implementation Proposals

It is tempting to implement all STI COM interfaces with generic implementations, but that is impossible. The drivers for scanners rely on the vendor-supplied minidriver which implements `IStiUSD` interface. This minidriver interprets command to particular device, therefore it is impossible to replace this minidriver with generic implementation.

Vendor minidrivers must be loaded and here comes the problem when the minidriver tries to open USB kernel module. The device path will parse as a VxD[7] in Wine and thus fail. Since VxD's do not implement `ReadFile()` and `WriteFile()`, or the correct range of IOCTL codes, they cannot be used to implement Wine's own USB module. `DeviceIoControl()`, `ReadFile()` and `WriteFile()` need to be implemented using libusb or Linux specific `ioctl()`'s that work on the `[/proc]` filesystem. At present, `NtDeviceIoControlFile()` calls `CDROM_DeviceIoControl()` and `NtReadFile()`/`NtWriteFile()` calls `read()`/`write()` which is not very useful.[8]

Possible solutions are:

---

[7] VxD is Virtual Device Driver. It runs under the Windows 3.x, 9x and Me operating systems, and have access to the memory of the kernel and all running processes, as well as raw access to the hardware

[8] http://www.winehq.com/pipermail/wine-devel/2005-September/039905.html

Kernel Module

Implement a kernel module that would accept `read()`, `write()` and `ioctl()` calls, and deal with them like windows USB scanning kernel would.

Advantages:

Only `CreateFile()` and `NtDeviceIoControlFile()` implementation need to be changed.

Disadvantages:

Completely Linux-specific, this work is not aimed to portability so this is not a problem. Inconsistent ioctl codes with Linux kernel numbering (created driver can not be included in Linux kernel tree).

Problems porting to different Linux kernel versions (e.g. project SANE had a scanner kernel module in 2.4 kernels, but it was replaced with libusb in 2.6)

For every type of bus there is a need for separate kernel module (USB, SCSI, serial, infra red), in fact this is not an issue of this work, because the only focus is on the USB devices. Kernel module for particular bus has to be generic to support all types of USB device configurations, because every type of a scanner can have different USB endpoints in its configuration. For example, different endpoint for image transfers for scanners from different vendors can be used.

Dll Patching

Patch DLL imports for the minidriver so that `CreateFile()`, `ReadFile()`, `WriteFile()` and `DeviceIoControl()` get dynamically linked to alternative implementations that use libusb functions.

Advantages :

No changes to existing Wine code.

Disadvantages :

Wine does not support DLl patching.

Many versions of each function for USB, SCSI, etc.

Typed Handles

Modify handle to store handle type, or else function pointers to functions used on handles, like reading, writing and ioctl. Ntdll functions like `NtWriteFile()`, `NtReadFile()` and `NtDeviceIoControlFile()` should use the handle type to demultiplex the I/O request to the correct function. For example when `NtDeviceIoControlFile()` is called on a handle of type `HANDLE_USB_SCAN`, the function `UsbScanDeviceIoControl()` is called. Each handle would have also generic data pointer to associate internal data with handle.

Advantages :

Easy to add STI USB, and also SCSI, serial and infrared device types.

Easy to add other hardware support in general.

Disadvantages :

This might require changes to Wine server.

Wine community does not want to support device drivers in Wine.

9.1.5 STI Interface to USB Implementation

The decision was to implement STI interface to USB by using kernel module as described in 9.1.4. This implementation requires minimal changes to Wine architecture, in fact it is only addition of new features to existing Wine implementation.

9.1.6 Implementation of STI Architecture

STI architecture is designed as pull and push acquisition protocol as illustrated in figure 9.3. Pull protocol means that application requests an image from STI and the request goes all the way down to the scanner.

Push protocol means that an application can register with STI to handle scanner events. For example if button is pressed on scanner this event is propagated through STI and registered application is launched.

The decision is to implement only STI pull acquisition protocol. Push protocol cons are that events from scanner can be catched by Linux kernel scanner module, but there is a problem how to propagate this event to Wine. Wine has the information about registered applications stored in user's Registry that are not accessible to kernel module.

9.2 TWAIN Implementation

TWAIN working group provides TWAIN Development Kit package that was used during implementation. In this package there are sources of TWAIN testing application that was used for testing purposes for TWAIN elements cooperation - DataSourceManager and DataSources. TWAIN Development Kit sources where with small changes compiled by Wine compiler. This was achieved by following the winelib how-to.

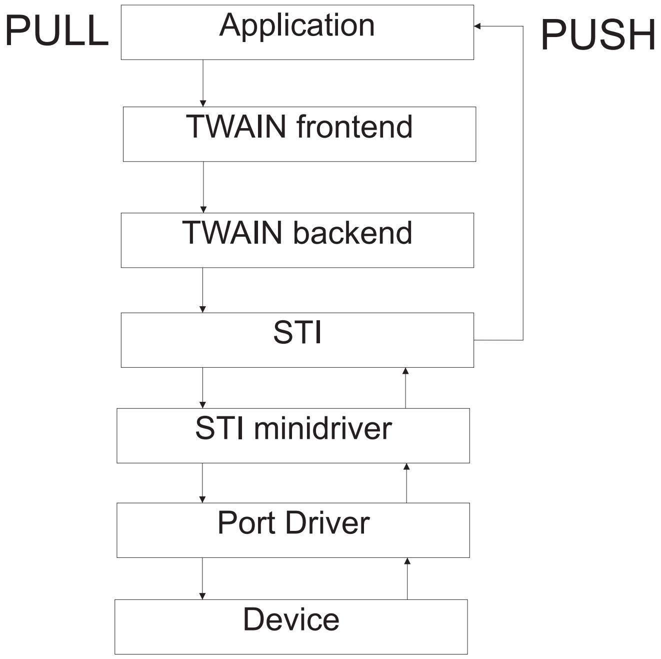

Figure 9.3: STI push vs pull model

9.3 Implementation Issues

9.3.1 Supported Scanners by USB Scanner Kernel Driver

Linux USB scanner kernel driver is implemented not to drive all USB devices probed. If a new type of scanner has to be driven by this driver then function `probe()` has to be updated to support new vendors and/or devices. Following change is needed. Find following `if` command in `[usbscan.c]`:

```
if (dev->udev->descriptor.idVendor != 0x05DA) {
    retval = -ENODEV;
    goto error;
}
```

Update it to enable driving scanners from different vendors.

Driver can be modified to support configuration parameters vendor and device. This modification will remove need to recompile driver for different scanner devices.

9.3.2 Missing DLLs

In case Wine complains about missing DLL, one must locate it and then needs to make sure Wine is able to use it. DLLs usually get loaded in the following order:

- The directory the program was started from
- The current directory
- The Windows system directory
- The Windows directory
- The PATH variable directories

There was a problem with DLLs installed by DataSource during testing. They are usually installed into [$(WINDOWS_DIRECTORY)\twain_32\DATASOURCE_DIR]. This directory is not one of the directory types stated above. The optimal solution is to add `DATASOURCE_DIR` into `PATH` variable to solve the problem.

9.3.3 Wine's Problem with GUIDs

Used Wine implementation had problem with braces ({}) in GUIDs, when searching the Registry. The workaround for this issue is to duplicate Registry entry for {GUID} with new name GUID. Example of usage is available in section 9.4

9.4 Installed Scanner Driver Example

Following example will illustrate minimum set of Registry entries that need to be created for every scanner. This example will be for Microtek scanner where Vendor ID is 05da and Product ID will be 009a.

Key HKEY_LOCAL_MACHINE\System\CurrentControlSet\Enum\USB\Vid_05da&Pid_009a\5&23a18cd&0&3 must contain String values:

- ClassGUID = {6bdd1fc6-810f-11d0-bec7-08002be2092f}
- Driver = {6bdd1fc6-810f-11d0-bec7-08002be2092f}\0001

Key HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\{6bdd1fc6-810f-11d0-bec7-08002be2092f}\0001 and its duplicate HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\6bdd1fc6-810f-11d0-bec7-08002be2092f\0001 must contain values

- Capabilities = 00000003
- CreateFileName = \\.\\$USBSCAN$0
- DeviceSubType = 00000000
- DeviceType = 00000001
- DriverDesc = Microtek SlimScan C6u
- FriendlyName = Microtek SlimScan C6u
- HardwareConfig = 00000000
- PropertyPages = 00000000
- Vendor = Microtek

Note that numeric values Capabilities, DeviceSubType and DeviceType must have the same values for every scanner.

Key HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\DeviceClasses\{6bdd1fc6-810f-11d0-bec7-08002be2092f}\##?#USB#Vid_05da&Pid_009a#5&23a18cd&0&0#{6bdd1fc6-810f-11d0-bec7-08002be2092f} must contain String value

- DeviceInstance = USB\\$Vid$_05$da$&$Pid$_009$a$#5&23$a18cd$&0&3

Key HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\DeviceClasses\{6bdd1fc6-810f-11d0-bec7-08002be2092f}\##?#USB#Vid_05da&Pid_009a#5&23a18cd&0&0#{6bdd1fc6-810f-11d0-bec7-08002be2092f}\# must contain String value

- SymbolicLink = \\?\USB#Vid_05da&Pid_009a#5&23a18cd&0&3#{6bdd1fc6-810f-11d0-bec7-08002be2092f}

Chapter 10

Cooperation with SANE

Over the years SANE became standard scanning architecture in Linux world. It has very flexible architecture, stable and tested implementation and it offers a lot of implemented Linux scanning applications that can connect to standardized SANE interface. The next section will describe basics of SANE architecture and prepare the reader for design of cooperation SANE with TWAIN scanners.

10.1 SANE Introduction

SANE is an application programming interface that provides standardized access to any raster image scanner hardware. It is a universal scanner interface that allows writing just one driver for each scanner device instead of one driver for each scanner and scanning application. It allows easy implementation of the API while accommodating all features required by scanner hardware and applications. While SANE is primarily targeted at a UNIX environment, the standard has been carefully designed to make it possible to implement the API on virtually any hardware or operating system.

Scanning application that uses the SANE interface is called a SANE frontend. A driver that implements the SANE interface is called a SANE backend. A meta backend manages one or more other backends. Note that a meta backend is frontend and a backend at the same time. It is a frontend from the viewpoint of the backends that it manages and a backend from the viewpoint of the frontends that access it.

Accessing a raster scanner device typically consists of two phases:

- In the first phase various controls of the scanner need to be setup or queried.
- In the second phase one or more images are acquired.

Since the device controls are widely different from device to device, SANE is designed to abstract each device control into a SANE option. An option is a self-describing name-value pair. The backend simply provides a list of SANE options that describe all the controls available in the device. It is not concerned with the presentation, because of self-describing options.

There are three possibilities how SANE frontend connects to a SANE backend.

- Static linking - a SANE backend may be linked directly into a frontend. This is the simplest method of attaching to a backend. It is limited in functionality since the available devices are limited to the ones for which support has been linked in when the frontend was built.
- Dynamic linking - in this case, a frontend is linked against any shared library that implements SANE backend. It is possible to switch the backend by installing appropriate backend dynamic library. Dynamic linking makes it easy to implement a meta backend that loads other backends on demand. This is a powerful mechanism since it allows adding new backends merely by installing a shared library and updating a configuration file.
- Network connection - It is a way how to attach a scanner by using the network to connect to a backend on a remote machine. This makes it possible to scan images from any host, as long there is a network connection to that host and if the user has permission to access the scanner.

It is possible to combine these solutions to provide a hierarchy of SANE backends.

10.1.1 Image Data Format

The most important aspect of an image acquisition system is how image data are represented. The SANE defines a simple representation that is sufficient for majority of applications and devices.

A SANE image is a rectangular area. The rectangular area is subdivided into a number of rows and columns. At the intersection of each row and column is a pixel. A pixel consists of one or more sample values. Each sample value represents one channel.

The SANE API transmits an image as a sequence of frames. Each frame covers the same rectangular area of the entire image, but may contain only a subset of the channels in the final image. For example, a red/green/blue image could either be transmitted as a single frame that contains the sample values for all three channels or it could be transmitted as a sequence of three frames.

Sample values in a frame are transmitted row by row and each row is transmitted from left-most to right-most column. The left-to-right, top-to-bottom transmission order applies when the image is viewed in its normal orientation. If a frame contains multiple channels, then the channels are transmitted interleaved.

10.1.2 SANE API

The SANE standard is expected to evolve over time. Whenever a change to the SANE standard is made that may render an existing frontend or backend incompatible with the new standard, the major version number must be increased.

SANE version control also includes a minor version number and a build revision. Control of these numbers remains on the implementer of a backend. [9]

Data Types

Description of standard SANE data types can be found in [4]. In this section we will describe essential types only.

Device Descriptor Type

Each SANE device is represented by a structure of type `SANE_Device`. The structure provides the unique name of the scanner. This unique name is passed in a call to `sane_open()`. The format of this name is completely up to the backend. The only constraints are that the name is unique among all devices supported by the backend. The remaining members in the device structure provide additional information on the device corresponding to the unique name.

Scanner Handle Type

Access to a scanner is provided through an opaque type called `SANE_Handle`. While this type is declared to be a void pointer, an application must not attempt to interpret its value.

Status Type

Most SANE operations return a value of type `SANE_Status` to indicate the completion status of the operation.

Option Descriptor Type

Options are used to control all aspects of device operation. Options are described by `SANE_Option_Descriptor`. Thus, a frontend can control a scanner abstractly, without requiring knowledge as to what the purpose of any given option is. A scanner backend can describe its controls without requiring knowledge of how the frontend operates. Option `name` is a string that uniquely identifies the option. Option `title`

---

[9] SANE Standard Version 1.04, 2006, p. 13

is a single-line string that can be used by the frontend. Option `desc` is a long string that can be used as a help text to describe the option. Option `value` type specifies the type of the option value. The possible values are defined by `SANE_Value_Type`. Option value unit specifies what the physical unit of the option value is. The possible values are defined by type `SANE_Unit`. Option value size specifies the size of the option value. This member has a slightly different interpretation depending on the type of the option value. Option `capabilities` describe what capabilities the option possess. This is a bitset that is formed of the capabilities defined by SANE. Option value constraints are used to constrain the values that an option can take.

While most backend options are completely self-describing, there are cases where a user interface might want to handle certain options in a special way. These are

- option number count - is option number zero with empty string as its name. This option specifies the total number of options available for a given device.
- scan resolution option - is used to select the resolution at which an image should be acquired.
- preview mode option - is used by a frontend to inform the backend when image acquisition should be optimized for speed, rather than quality.
- scan area options - are represented by four options that define the scan area. The scan area is defined by two points that specify the top-left and the bottom-right corners.

Code Flow

The code flow for the SANE API is illustrated in Figure 10.1. Detailed description of SANE API can be found in [4].

Functions `sane_init()` and `sane_exit()` initialize and exit the backend, respectively. All other calls must be performed after initialization and before exiting the backend. Function `sane_get_devices()` can be called any time after `sane_init()` has been called. It returns the list of the devices that are known at the time of the call. This list may change over time since some devices may be turned on or off. Once a device has been chosen, it is opened using a call to `sane_open()`. Multiple devices can be open at any time given.

An open device can be setup through the corresponding device handle using functions `sane_get_option_descriptor()` and `sane_control_option()`. While setting up a device, obtaining option descriptors and setting and reading option values can be mixed freely. The device handle can be put in blocking or non-blocking mode by a call to `sane_set_io_mode()`. Devices are required to support blocking mode. If device does not support non-blocking mode it returns status unsupported.

After the device is setup properly, image acquisition can be started by a call to `sane_start()`. The backend calculates the exact image parameters at this point. So future calls to `sane_get_parameters()` will return the exact values, rather than estimates. Whether the physical image acquisition starts at this point or during the first call to `sane_read()` is unspecified by the SANE API.

Image data are collected by repeatedly calling `sane_read()`. Eventually, this function will return an end-of-file status. This indicates the end of the current frame. If the frontend expects additional frames, it can `call sane_start()` again. Once all desired frames have been acquired, function `sane_cancel()` must be called. This operation can also be called at any other time to cancel a pending operation.

Having completed the usage of the device, the handle should be closed by a call to `sane_close()`. Finally, before exiting the application, function `sane_exit()` must be called.

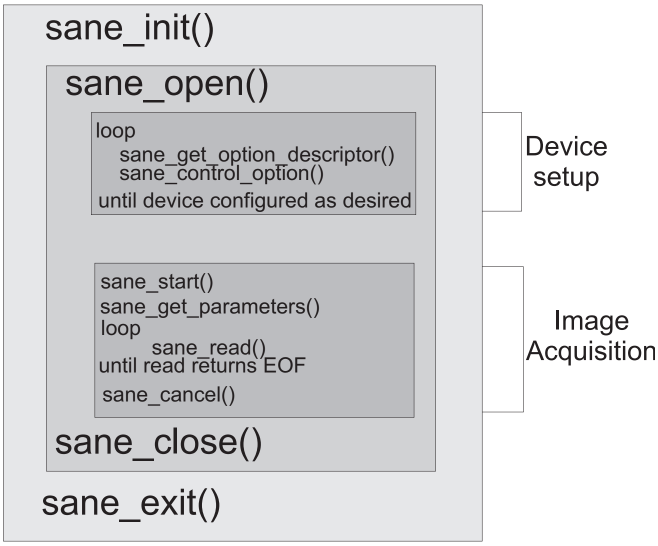

Figure 10.1: SANE code flow

Network Protocol

The SANE network protocol is a client/server style remote procedure call protocol. The SANE protocol can be run across any transport protocol that provides a reliable data delivery. The data transferred from the client to the server is comprised of the RPC code, followed by arguments for RPC call. The format of the server's answer depends on called procedure. The details about encoding of data types and format of requests and responses can be found in [4].

10.2 Design of SANE Cooperation with TWAIN

Our goal was to make TWAIN scanners work with SANE. The optimal solution was to implement some kind of wrapper around the TWAIN drivers that will be able to process SANE requests through network.

SANE design goals reflect our requirements that we want to achieve. SANE is designed to run on any UNIX platform, even it is virtually possible that it will run on any hardware or operating system. The implementation is system dependent as little as possible. The SANE design enables us to reuse a lot of its implementation on Windows platform. On Windows side we needed implement SANE backend that will connect to TWAIN DataSource Manager and TWAIN DataSource.

SANE also supports network connectivity that reflects our next requirement. SANE implementation provides SANE daemon which is special type of SANE meta backend. SANE daemon implementation was ported with required changes to Windows platform.

On server side, in daemon, there were used SANE implementation and also in new implementation (of the twain-sane backend) we used SANE helper functions whenever possible. On client side there were no changes in implementation required. The only change was needed in configuration of dll meta backend to use also the net backend. In the net backend configuration we only added IP address of host where Windows version of SANE daemon runs. Proposed solution is illustrated in figure 10.2.

Many of the implementation efforts were focused on the communication with TWAIN DataSource Manager, TWAIN DataSource and to map the TWAIN states to SANE states. Figure 10.3 illustrates mapping of TWAIN states on SANE states. Transitions of SANE states are in one direction only from top to bottom. Transitions between TWAIN states are shown in chapter 8. These transitions can occur in both directions between two adjacent states.

In `sane_init` function we will load and open TWAIN DataSource Manager. If

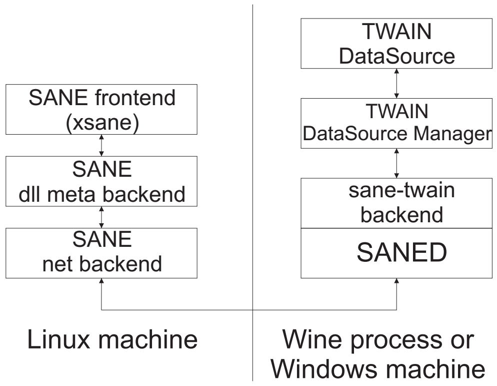

Figure 10.2: SANE - TWAIN cooperation

this action is successful then the first TWAIN DataSource is opened. Here we need to note that TWAIN does not separate the user-interface from the driver of a device, thus here we can have a problem when DataSource is opened and no scanner was found, then the GUI on server for DataSource is displayed. This GUI blocks until it is handled properly and the client side will block, too.

Proposed implementation will load the first DataSource found by DataSource Manager. In the future this implementation can be extended to support more DataSources installed on a computer, but this feature was not our aim. This can be achieved by iteration through all DataSources and expose them to SANE as scanners, so the client can choose one of them.

There is an objective purpose to open DataSource in this function. Before transition to another state we need to know the attached scanner that is driven by particular DataSource. This scanner information is later used by frontend in call to `sane_open` function.

When `sane_open` is called no transition in TWAIN states will occur. We will remain in TWAIN DataSource open state.

Frontend can now retrieve values for scanner's options. While TWAIN DataSource is in state opened its option values can be also negotiated. Thus the only possible mapping between SANE and TWAIN states here is that `sane_open` maps to Source open.

After all values are negotiated by SANE frontend we are ready for image aquisition. Call to `sane_start` will move DataSource to state enabled and wait for DataSource till it is in a state ready to transfer. Then it will start acquisition of the whole image. Acquired image is stored in memory and the DataSource will move to state DataSource enabled.

Bitmap stored in memory is in Microsoft's DIB format. This bitmap can not be sequently read by SANE. We need to make a few adjustments before providing data to SANE. DIB stores rows upside down. That means that the upmost row which appears on the screen actually is the lowest row stored in the bitmap. DIB uses row padding to adjust row length to a multiple of dword size. To SANE we need to provide image data beginning from first byte of first line without padding.

Now SANE frontend will repeatedly call function `sane_read` to retrieve image data. This data are send back from memory. When there are no more image data available EOF status is returned. After that `sane_cancel` is called and the TWAIN DataSource is disabled. After this call TWAIN DataSource is back in state open.

Call to `sane_close` will not change TWAIN state. It will only free structures used by our TWAIN scanner.

When sane backend will not be needed then function `sane_exit` is called. Here we will tidy up the whole TWAIN: DataSource is closed, DataSource Manager is closed and unloaded.

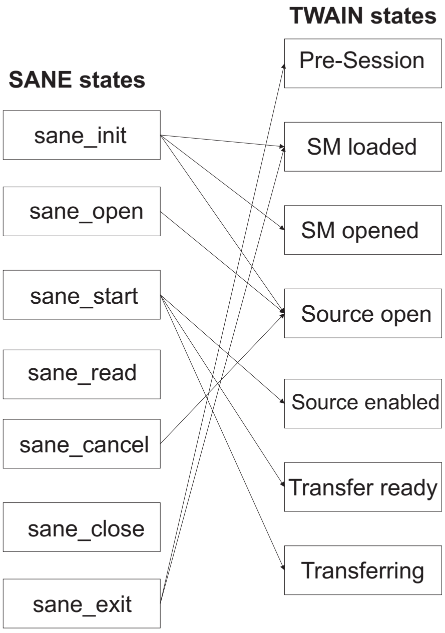

Figure 10.3: Map TWAIN states to SANE states

Chapter 11

How To Put it all Together

All files mentioned in this chapter can be found on the CD attached to this work, if not stated otherwise.

11.1 Scanner Kernel Driver

Copy directory `[usbscan]` with source code for kernel driver from CD.

11.1.1 Customization of the Scanner Driver

1. When we connect scanner for the first time we can use application `[usbview]` to inspect scanner properties. This application can be downloaded from `[http://usbview.sourceforge.net]`.
2. In `[usbview]`, on the left side we can see plugged USB devices. The red ones are devices for which no suitable driver was found. The black ones are with a driver loaded and ready to use.
3. Our scanner has to be in a red state. If it is not, then unload found driver and disable the driver loading for this particular device.
4. Now scanner is in a red state, when we click on it we can see its properties in the right panel. Find Vendor and/or Device properties.
5. Configure kernel driver to drive our scanner. Configuration is described in 9.3.1, where properties Vendor and/or Device are used.

11.1.2 Compilation, Loading and Unloading

Compile kernel driver with command `[make]` typed in `[usbscan]` directory.

Load driver into memory with command `[insmod]`. Example of the command use: `[/sbin/insmod /path/to/driver/usbscan.ko]`. When the driver is successfully loaded and scanner is plugged in, then the scanner is in program `[usbview]` in black state.

Change access permissions to files `[USBSCAN0]`, $[USBSCAN0\backslash 0]$, $[USBSCAN0\backslash 1]$, $[USBSCAN0\backslash 2]$ in `[/dev]` dirctory, if needed.

Unloading the driver from kernel can be done by command `[rmmod]`. Example of command use: `[/sbin/rmmod usbscan]`.

Kernel driver is implemented to print debug messages. These messages can be shown by command `[dmesg]`.

11.2 Still Image Subsystem in Wine

1. Download Wine source code from `[http://www.winehq.org]` and prepare it for compilation.
2. Add directories `[sti]`, `[sti_ci]` to wine build tree into `[dlls]` directory. Merge file `[kernel/file.c.change]` with file found in `[WINE_SRC_DIR/dlls/kernel/file.c]`. Mentioned directories and files can be found on CD attached to this work.
3. Modify Wine's `[configure.ac]` file to reflect new DLL components. In this file change value of variable `AC_CONFIG_FILES`. Add following lines to its value `dlls/sti/Makefile`, `dlls/sti_ci/Makefile`.
4. Adjust code of the TWAIN DataSource Manager as mentioned in section 9.1.3
5. Compile and install wine.
6. Now Wine is prepared for scanner installation. Run install program for your scanner driver and follow steps stated by scanner vendor.
7. Now create symlink called `[sti-twain.ds]` in directory `[c://windows/system32]`. This symlink should point to DataSource installed in previous step.

If scanner installation program fails, probably it was because of lack of Wine's implementation of SetupAPI. In this case you can try to install scanner on Windows machine. Then copy directory with installed TWAIN scanner driver to Linux machine and copy registry entries mentioned in chapter 7 and also in section 9.4. Installed TWAIN scanner driver should be in directory `[WINDOWS_HOME/Twain_32]`.

11.3 SANE Scanning

To enable SANE scanning configuration is required on server and client side.

11.3.1 Configure Server Side

Requirements are that scanner is installed and configured properly. Steps required to install and configure the scanner are described in the previous sections.

1. Copy compiled version of `[saned-twain]` directory from CD.
2. Edit file `[saned.conf]`. Change entries of client IPs from which clients are permitted to access scanner.
3. Start `[saned-twain]` daemon. Daemon can be started from shell with command `[wine SANED-TWAIN.EXE]`. Daemon can run in normal or debug mode. To enable debug mode export environment variable `SANE_DEBUG_TWAIN` with value from 0 to 255. Greater value means more debug messages. For example command `[export SANE_DEBUG_TWAIN=255]` will set debug variable for saned-twain to maximum sensitivity.

Compilation of Saned-twain

Saned-twain is project created in Bloodshed Dev-C++ v4.9.9.2 [10]. To edit/compile this project it is necessary to open its project's file `[SANED-TWAIN.dev]` in Dev-C++. To change application type (GUI/Console) go to project options submenu of the project menu.

Project can be compiled from menu Execute submenu Compile.

11.3.2 Configure Client Side

Requirement is that client side has compiled SANE with backends dll and net.
The following configuration is needed.

1. In file `[dll.conf]` enable net backend. In file `[net.conf]` configure servers running `[saned]`, that the client should connect to. Mentioned configuration files can be found in directory `[/usr/local/etc/sane.d/]`, in version of SANE compiled from source code.
2. Run any SANE frontend and try the scanner. Frontend tested with this project are `[xsane]`, `[scanimage]` and `[xscanimage]`.

---

[10] Dev-C++ homepage is http://www.bloodshed.net

11.4 Known Problems

Saned-twain daemon has to be compiled as GUI application if it is run in Wine. But then if it runs in debug mode, all debug messages will be printed after daemon finishes. When daemon will run under Windows it can be compiled as Console application to behave normally in the debug mode.

Chapter 12

Conclusion

Our goal was to design and implement environment around Windows scanner drivers that will enable scanner drivers to drive the scanner. We have successfully implemented scanning subsystem stack that consists of low-level Windows API - STI and high-level API - TWAIN. We provide compatible interface for Windows USB scanner drivers that allow them access scanner hardware. Finally, we achieved to incorporate Windows scanner architecture into Linux scanning architecture called SANE. Scanning tests were successfully made through SANE frontends.

Windows scanning architecture was implemented with the help of Wine project, which is a free implementation of Win32 API. Our improved Wine provides environment for scanning subsystem and also for SANE network daemon that exposes Windows TWAIN USB scanners to SANE. The side effect of our design is that we can expose TWAIN compatible scanners, installed on Windows machines, to UNIX machines that run SANE configured to access network scanners.

Some restrictions were made upon our implementation.
We support only one USB scanner device connected to the system. That means we can handle only one device that is connected to the Linux machines and used with our scanning subsystem and also only one scanner that is accessible through our SANE daemon.
Our USB kernel driver will not automatically drive every USB device connected to the system, but this is only configuration restriction. Configuration can be changed in kernel driver code. Steps needed for reconfiguration are described in this work.

Continuation of this work is possible in two areas. First is that Wine lacks implementation of SetupAPI that is needed to install scanners from some vendors. We have implemented some of the SetupAPI to be able to install some scanners, but there is still a lot of work to be done. Second possible area of interest is to implement subsystem for another class of USB devices, for example printers or cameras.

Bibliography

[1] *Microsoft Portable Executable and Common Object File Format Specification, Revision 8.* Microsoft Corporation, May 2006.

[2] *Wine Developer's Guide.* Wine Community, March 2007.

[3] *TWAIN Specification.* TWAIN Working Group Committee, 1.9 edition, January 2000.

[4] *SANE Standard Version 1.04.* January 2006.

[5] *Microsoft Driver Development Kit.* Microsoft Corporation, 2006.

[6] David A. Solomon Mark E. Russinovich. *Microsoft Windows Internals: Microsoft Windows Server.* Microsoft Press, fourth edition, December 2004.

[7] Detlef Fliegl. *Programming Guide for Linux USB Device Drivers.* 1.32 edition, December 2000.

[8] Greg Kroah-hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers.* O'Reilly, third edition, January 2005.

[9] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel.* O'Reilly, 2nd edition, 2002.

[10] *Wine User's Guide.* Wine Community, March 2007.

[11] *Winelib User's Guide.* Wine Community, March 2007.

[12] *SCSI Command Set for Microtek scanner, Revision 2.2.4.* Microtek Corporation, 1997.

[13] http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx.

[14] http://msdn.microsoft.com/msdnmag/issues/02/02/PE2/.

[15] http://msdn.microsoft.com/library/en-us/dndebug/html/msdn_peeringpe.asp.

[16] http://www.winehq.com/pipermail/wine-devel/2005-September/039905.html.

[17] http://www.captain.at/programming/kernel-2.6.

Glossary

- `API` - Application Programming Interface
- `COFF` - Common Object File Format
- `DIID` - Combination of Device ID and Instance ID in Windows OS
- `DLL` - Dynamic Link Library
- `FD` - File Descriptor
- `GUID` - Global Unique Identifier
- `HAL` - Hardware Abstraction Layer
- `I/O` - Input/Output
- `INF` - Information file
- `IRP` - I/O Request Packet
- `NT` - New Technology
- `OS` - Operating System
- `PnP` - Plug and Play
- `RVA` - Relative Virtual Address
- `SANE` - Scanner Access Now Easy
- `STI` - Still Image subsystem
- `TWAIN` - Standard for acquiring images from scanners
- `URB` - USB Request Block
- `USB` - Universal Serial Bus
- `UUID` - Universally Unique Identifier

- `VA` - Virtual Address
- `VxD` - Virtual Device Driver
- `WDM` - Windows Driver Model
- `Wine` - Wine Is Not an Emulator

Abstract in Slovak Language

Práca sa zaoberá vytvorením vhodného prostredia pre Windows USB ovládače pre skenery v prostredí operačného systému Linux. Naším výsledkom je použitie Windows USB ovládačov pre skenery v prostredí Linuxu. Ďalším výsledkom je zapracovanie implementovanej Windows skenovacej architektúry do Linux SANE skenovacej architektúry.

Práca začína vysvetlením modelu Windows ovládačov a Windows skenovacej architektúry. Pokračuje modelom Linux ovládačov a implementáciou user space Win32 API v prostredí Linuxu nazvanou Wine.

Jadrom práce je popis návrhu a návrhárskych rozhodnutí implementácie Windows skenovacej architektúry v prostredí Linuxu. Nakoniec sa venujeme opisu detailov prepojenia implementovanej skenovacej Windows architektúry s Linux skenovacím subsystémom.

Poskytli sme aj návod ako spojiť všetky časti implementácie a nastaviť skener aby fungoval v prostredí Linuxu s Windows ovládačom.

Kľúčové slová: ovládač, skener, Linux, Windows