Matthew Johnson

CCID Smart-Card Library

Computer Science Tripos, Part II

Trinity Hall

2004

Proforma

Name:	**Matthew Johnson**
College:	**Trinity Hall**
Project Title:	**CCID Smart-Card Library**
Examination:	**Computer Science Tripos, Part II. 2004**
Word Count:	**9185** [1]
Project Originator:	Dr. M. Kuhn
Supervisor:	S. J. Murdoch

Original Aims of the Project

The original aim of this project was to produce a user space library for accessing the features of ISO 7816[6]-based smart-cards via USB readers. This library would be available to programs which use these cards, to provide an easy and consistent interface to their features, and would also perform secure multiplexing between programs and card readers.

Work Completed

I have completed a framework for a user space library for accessing ISO 7816[6]-based smart-cards, and have implemented enough code within the framework to demonstrate its operation. A description of all that has been completed appears in this dissertation, and selected excerpts of code form the appendices.

Special Difficulties

None.

[1] This word count was computed by `ps2ascii dissertation.ps | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Matthew Johnson of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1 Introduction **1**

2 Preparation **3**
- 2.1 Familiarisation with Tools . . . 3
- 2.2 Requirements Analysis . . . 3
- 2.3 Specifications . . . 4
  - 2.3.1 CCID Class-spec . . . 4
  - 2.3.2 ISO 7816 . . . 5
  - 2.3.3 APDU Syntax . . . 6
- 2.4 Architecture . . . 8
- 2.5 Choice of Tools . . . 8

3 Implementation **11**
- 3.1 Architecture . . . 11
  - 3.1.1 Security Implications . . . 13
- 3.2 Library . . . 13
  - 3.2.1 Example APDU Methods . . . 14
  - 3.2.2 Implementation of Library Functions . . . 14
  - 3.2.3 Library Documentation . . . 15
  - 3.2.4 Selecting a Card . . . 15
- 3.3 Daemon . . . 15
  - 3.3.1 UNIX Process Management . . . 18
  - 3.3.2 Socket Communication . . . 18
  - 3.3.3 Managing USB devices . . . 19
  - 3.3.4 USB Protocols . . . 20
- 3.4 Library-Daemon Protocol . . . 22

4 Evaluation **25**
- 4.1 Serviceability . . . 25
  - 4.1.1 Library API . . . 25

4.1.2 Application-Daemon Protocol 26
4.1.3 USB Code 26
4.1.4 T=0 and APDU 26
4.2 Simplicity 26
4.3 Stability 27
4.4 Security 27
4.5 Additional Features 29
4.5.1 Interrupt-based events 29
4.6 Testing 29
4.7 Summary 30

5 Conclusions **31**
5.1 Achievements 31
5.2 Future Work 31

Bibliography **33**

Appendices **35**

Project Proposal **51**

List of Figures

2.1 Example File Structure (taken from ISO 7816 [2]) 6
2.2 Different File Types (taken from ISO 7816 [2]) 7

3.1 Architecture Overview Diagram 12
3.2 Example Library Interfaces 14
3.3 Example Output from CWEB 16
3.4 CWEB Source Code for Figure 3.3 17
3.5 APDU Command Structure (taken from ISO 7816 [2]) 21
3.6 Structures for Storing APDU Commands 23

4.1 Example Application Code 27

Chapter 1

Introduction

Until recently, the smart-card industry hasn't had a consistent set of standards or specifications for their cards. This has led to a large number of proprietary solutions, all of which are incompatible and require different device drivers and interfacing software. This has made it very difficult to write drivers to support them and also difficult to produce applications which can work seamlessly across many different types of card. The same features may not be supported, and the APIs for accessing these features will be different.

There are projects available that attempt to solve this. The MuscleCard Project[1] for the GNU/Linux operating system is one of these which has been quite heavily developed. However, they all suffer from the problem that in having to support all the available types of smart-card, they tend to be quite bloated and difficult to use.

There are several standards governing the design of and interface to smart-cards. ISO Specification 7816[6] describes how to build smart-cards from the electrical and physical level, to the protocols which should be used on top, and has a consistent framework for extending the protocols with proprietary commands. This specification is endorsed by people like the Eurocard-Mastercard-Visa (EMV) group who have designed the smart-cards built into most credit and debit cards, and is likely to be used for most if not all new smart-cards developed.

A second problem with the majority of smart-card readers is with their interface to the computer. Most of them have used either the parallel or serial busses to connect to the computer, or even a custom connection direct onto the motherboard. This raises yet more problems as those IO ports do not easily multiplex multiple devices seamlessly, and are in short supply.

[1] http://www.linuxnet.com/musclecard/

This has all been changed with the popularisation of the Universal Serial Bus (USB)[5]. USB has allowed machines to have as many devices attached as necessary, and with a consistent cross-platform interface. Due to its extensibility and improved speed, USB is rapidly replacing all the legacy IO connectors, and is likely to be the only method that needs to be supported in the near future. The USB Chip/Smart-Card Interface Device (CCID) Class specification gives a protocol for accessing these devices over the Universal Serial Bus[7].

With both these things in mind, Dr Kuhn and I decided that a clean implementation of the ISO and CCID specifications to interface with USB-based smart-card readers was necessary. I have primarily developed this for the GNU/Linux operating system, but since I have written code entirely in user-space, the only platform-specificity is in the USB interface, which is being provided by LibUSB[3] – which supports a wide range of UNIX-like operating systems.

Finally, a feature which has been entirely missing from the previous generation of drivers is multiplexing of applications talking to cards. This project provides a system where multiple smart-cards can be in use by multiple applications at any one time, and where the driver system manages the insertion/removal of cards, and enforces the separation of applications and the cards they are accessing.

Chapter 2

Preparation

This section summarises the work undertaken prior to the implementation of the project. Current systems in this area are discussed and I briefly outline the requirements of my system. I also list the tools used in the production of the project.

2.1 Familiarisation with Tools

The tools I decided to use for this project are listed in Section 2.5. Several of them were either new to me, or I had not used them for some time. Therefore, before starting to work on the project I had to do some preliminary work to familiarise myself with them. I have a fair amount of experience with LaTeX and C++, but not with CWEB, and I had not used C for several years. I wrote a selection of programs to familiarise myself with using CWEB. These programs were also used to investigate how the C standard libraries worked to handle UNIX sockets and access to USB devices, which are both primitives that I used during the project. This took a few weeks of my initial preparation time.

2.2 Requirements Analysis

Before starting to write the project I had several meetings with my supervisor, as well as Dr Kuhn, who suggested the project, to discuss what they thought would be useful in this area. I also investigated what was currently available in this area, and read through the ISO specifications so that I had an overview of how they all worked.

The basic functionality that the project needed to provide is to allow the applications to access all the high-level features of the card which are specified in [6]. There are several high-level protocols which are specified and different cards may implement a different selection of them (see Section 2.3.2 for more details). Ideally the project should be able to support all of these features, and be able to support the use of proprietary vendor-specific extensions.

In addition to the above, to make this project stand out from existing systems the features should be presented with as much abstraction as possible. Therefore, I wanted to provide the access to features as function calls in a high-level language, and hide as many of the implementation details as possible. This is relevant, since the underlying wire-protocol has two different variants (mentioned in Section 2.3.2), which do not need to be exposed to the application.

Finally, we wanted to provide an additional security layer which allowed multiple applications to be running accessing multiple smart-cards, without having to provide the applications with direct access to the devices concerned so that it is possible to ensure that the security policy isn't broken. This was to be done by splitting the driver into several sections, with only part of the driver having privileged access to the devices.

2.3 Specifications

There are two main sets of specifications that cover this area. The first is the USB CCID Class-Spec, which specifies a USB Class-ID for CCID-based smart-cards, and contains the information about how to encapsulate the higher-level protocols on the USB bus. The second is an ISO standard covering the application-level protocols which are common regardless of the bus used to connect the device.

2.3.1 CCID Class-spec

The USB Chip/Smart-Card Interface Device Class-Spec[7] allocates a new USB Class-ID of `0x0B`. This class identifies all devices which are USB CCID readers. This class ID can be detected by the Linux operating system and can notify programs when devices of this type are inserted or removed. The class specification also defines interfaces for getting extended information about the capabilities of a device, such as supported voltages and protocols. It also, importantly, includes the information about how to encode the ISO

specifications onto the USB bus, and how to power up the device to get the Answer To Reset.

The Specifications list three endpoints which a CCID reader may support: Bulk-In and Bulk-Out, and Interrupt. The first two carry the commands to the device and are used by the current projects attempting to implement drivers. The Interrupt endpoint is a feature which has only been implemented in the latest version of LibUSB and carries out-of-band data from the smart-card reader, allowing programs to be notified of events such as card insertion and removal.

Communicating with the card reader is done via a message / response structure. The messages defined in the class specification for accessing the card include powering up and down the card, getting the status of a slot on the reader, setting various parameters, and sending blocks of commands, such as an APDU command. There is also a separate message for secure (PIN-restricted) operations. I used these messages and the corresponding responses. This is investigated in more detail in Chapter 3 Section 3.3.4, and the code implementing can be seen in the complete source documentation[1].

Most of the options which are defined in here are not of interest to the application programmer. They specify how a particular card reader should be communicated with, and users of the library wish this to be completely transparent and work with any compliant reader device.

2.3.2 ISO 7816

The ISO 7816[6] specification outlines several different protocols for accessing the cards at various different levels, and specify what operations are valid at each level.

At the highest level is the interface that application programmers want to access. This is what is exposed to programmers via the library API. The most common interface, and the one I focussed on, is the Application Protocol Data Unit (APDU) protocol. This is based on operations on a structure which has file-system-like semantics, and contains read and write operations on these files as well as more specialised commands. A more complete description of this protocol is given below in section 2.3.3.

The other protocols include TPDU and an SQL-like syntax for accessing the smart-card like a database. TPDU is a similar protocol to APDU, but lower level and supported by some card which don't support APDU. I did not initially implementing these protocols, but the design of the system will keep in mind extensibility to these protocols as well. The specification defines methods to find out the capabilities of the various cards. These

will be exposed to the programmer, and used to verify that a particular command can be used on that card.

Those protocols are all application-level protocols which need to be exposed to the applications via the driver. The standard also defines lower level protocols which define how the cards communicate with the reader. Called `T=0` or `T=1`, the project handles these two internally, so that the application programmer does not need to specify which one is being used. The protocol in use is determined by the data sent as the answer to the reset command. Answer To Reset is an important part of all the protocols, and encodes a lot of data about the card.

At a lower level still the ISO standard also defines the electrical and physical characteristics of the cards, but that is the domain of the CCID reader, and not relevant to this project.

2.3.3 APDU Syntax

Files

The data on the smart-cards which support APDU is, at the Application Protocol level, organised in a file hierarchy, or tree structure. This starts with the Master File (MF) as the root node of the tree, and a binary tree below it of Dedicated Files (DFs). The Master File is a mandatory DF used as the root node of the tree. In addition, any of the DFs my have associated Elementary Files (EFs). There are two types of EF, Internal EFs contain data which is used by the processing units on the card and Working EFs are purely data storage for external programs. Figure 2.1 shows an example file layout.

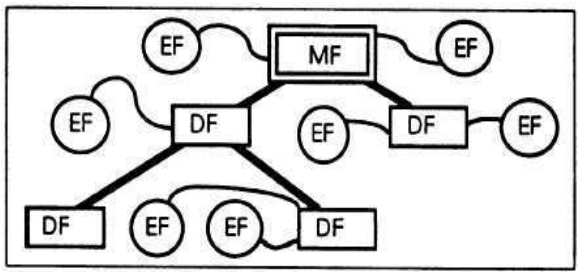

Figure 2.1: Example File Structure (taken from ISO 7816 [2])

APDU commands operate on a particular EF. This may be specified implicitly - the specification has the notion of a current file on which operations will be performed is no file is specified explicitly. Or, it may be specified explicitly in the command by file identifier. The file identifier is a

two byte number, which is unique among all DFs and EFs at a particular branch point in the tree. As with normal file structures, EFs may have the same file ID if they have different paths to them in the tree. The MF is always referred to by the reserved value `0x3F00`.

Elementary files can also have several different structures, although not all cards support all of the different structures. A transparent EF is a standard block-accessed file with no explicit internal structure. Record structure EFs are explicitly encoded as being made up from individual records (fixed or variable length) and which can be organised either linearly or cyclicly. There is the notion of a current record pointer which indicates the record currently being accessed. All of the record commands can either implicitly use the current record, or select the record relative to the start and end of the file, or the currently selected record. Figure 2.2 shows these different file access methods.

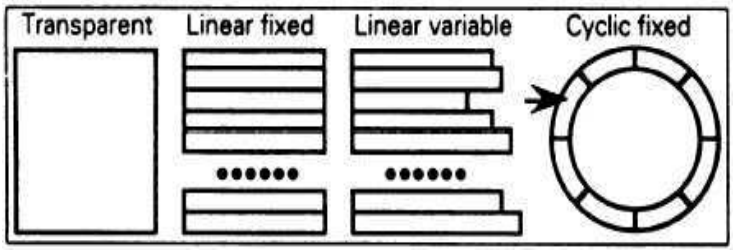

Figure 2.2: Different File Types (taken from ISO 7816 [2])

APDU Commands

APDU commands are structured as messages between the application and the smart-card. The application sends command messages to the device, which then sends a response back. The command message contains a command header which describes the command and its fixed length parameters. It and may also contain an attached variable length data item and the length of any expected response. The response header contains a status field, and may contain a variable length response.

There are four types of APDU command message. The simplest type is a 4-byte command header. with no attached data, and no expected response data. Type 2 and 3 either contain a 2 byte expected response length, or a 2 byte length and attached data field of length bytes as a command parameter. The final type contains both a data parameter and an expected response length. This final type is 8-bytes plus a variable length data field.

The exact coding of each command is given in more detail in Chapter 3 Section 3.3.4. The code for managing the encoding and decoding them is visible in Appendix B Section 81.

2.4 Architecture

The USB subsystem on GNU/Linux frequently uses a package called hotplug[1] to manage devices as they are inserted or removed. This involves a daemon which runs and constantly monitors the USB bus. When a new device is inserted into the machine hotplug tries to establish which modules are required to use it, and possibly launches external programs to deal with it.

I wanted to extend this architecture to the smart-cards and readers. This would allow programs to register themselves as handlers for particular smart-cards, and to be notified of the insertion and removal of smart-cards, and for my program to launch them if necessary.

2.5 Choice of Tools

This project is designed to be used by a lot of other programmers, and incorporated in a range of other programs, typically those written for the GNU/Linux Operating System. Traditionally programs for UNIX-like Operating Systems, particularly those such as device drivers, have been written using the C programming language, and to allow this project to be used in as wide a range of applications as necessary, I decided to use the most common language. Many of the programs which could benefit from this project are already in existence and in the most part use C.

For documentation purposes I decided to write the program using the CWEB System of Structured Documentation, by Knuth and Levy[8]. This documents and marks up the source using the TEX typesetting language. Documentation and comments are interspersed with the code throughout the source files, and later converted into pure TEX and pure C. This is compiled using the GNU Compiler Collection C compiler[2], using GNU `make` to manage the build process. The source files were edited using the Vi IMproved editor with C syntax hi-lighting.

Development was performed on a selection of machines, either my personal machines running Debian GNU/Linux, or Public Workstation Ma-

[1] http://linux-hotplug.sourceforge.net/
[2] http://gcc.gnu.org/

chines running Red Hat Linux and managed by the University Computing Service. The source was stored in the Perforce[3] Revision Control System on my personal server running Debian GNU/Linux, but nightly generation backups were taken and distributed to a number of other machines distributed over Cambridge, including the University Archive Server (Pelican) and the machines of the Student Run Computing Facility[4]. This process was automated using `logrotate` and a custom POSIX SH Shell script.

Perforce was chosen over systems such as CVS or RCS due to a number of improved features, including a very good graphical environment, and although I never had to use any of the roll-back features it was simple to use and made backups and developing on several machines very easy.

[3] http://www.perforce.com/
[4] http://www.srcf.ucam.org/

Chapter 3

Implementation

This chapter describes the implementation of a system to meet the requirements given in Chapter 2 Section 2.2. The architecture of the system is described, as are the protocols that were designed to interface between the various components of the system.

I will start by giving the architecture overview, and then give details of each section, and all the protocols, in more depth individually. The main components are the library which provides the user-level API and the system daemon, which provides all the communication with the hardware. The protocols used between the cards and the card readers and the daemon are given in the specification documents listed in Section 2.3, but I will elaborate further below. There is also a protocol needed between the library and the daemon, and that is a custom protocol designed for this project.

3.1 Architecture

As I mentioned in Section 2.2, I wanted to provide a system which could provide a secure layer of separation between several applications using different cards. This would have to apply even if the applications were running as different users and the cards were in different slots of the same card reader, or the same slot at different times. This suggested the use of a two level system, comprised of a privileged part, which would have direct access to the devices and would mediate application access, and an unprivileged part which would be part of the applications. The interactions between these can be seen in Figure 3.1. This figure illustrates the case of one application (marked in red) accessing two smart-cards, and the other application accessing a card in a shared reader.

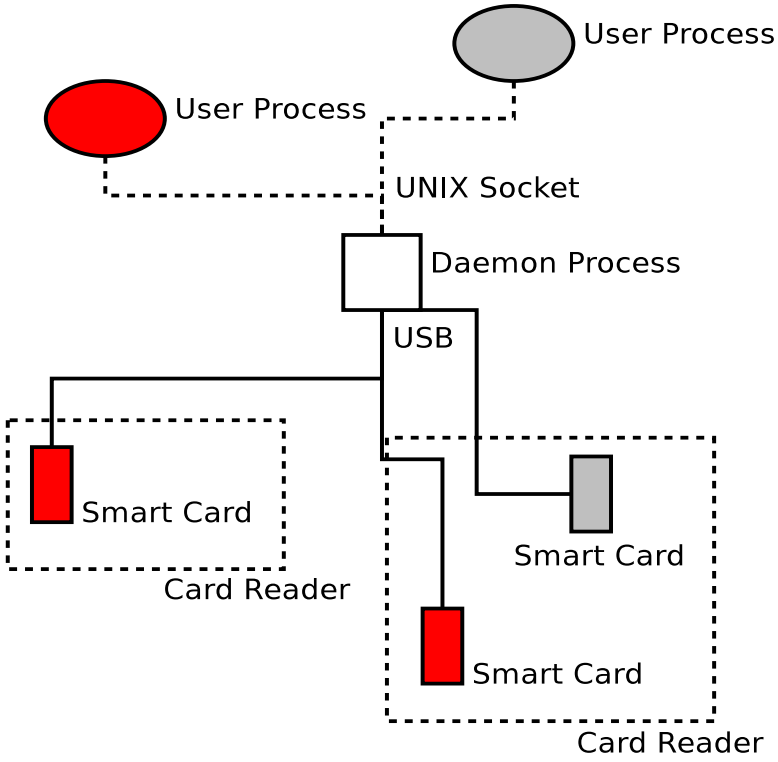

Figure 3.1: Architecture Overview Diagram

3.1.1 Security Implications

This project handles the communications between the applications and the smart-card, and is not trying to dictate their security policy or model. Also, the smart-cards themselves have a security policy which they enforce regarding access to secrets. Both the application and the smart-card assume that once they start communicating and have sent authentication tokens to the other, the channel is restricted to them. This is the only security policy which I am trying to enforce with this project. Therefore, security credentials merely need to be maintained over the course of the session, and a session must be completely separate from any other on the same card. How this is done can be seen in Section 3.3.2.

3.2 Library

As the functionality of the project is defined by the library API that is presented to programmers, it was natural to start by specifying this, and then building a system which would be able to support it.

The specifications of the system are to provide a simple interface to the programmer to access all of the features that the card provides in the APDU[1] and TPDU interfaces. Due to time constraints I decided to start with just the APDU functionality, but designed the system such that TPDUs could be easily added.

APDUs are easily classified into several sections, such as Binary- or Record-based accesses of files on the card. In an object-oriented language such as Java or C++ I might have implemented this using objects for each type of APDU, and overloading the various methods to handle each one differently. Since I was restricted to using the most common programming language for the target operating systems (C) which lacks such high-level concepts, I could not do so. I originally planned to have a generic method which would take a parameter to govern which APDU call would be made. Unfortunately there is such a large diversity in parameters needed for APDU calls, that this would not work. Another possibility would be to pass in a structure containing all the parameters, which could vary for each type of APDU. I discarded this as being too complex and unwieldy to be used regularly in programs. Therefore, I decided to use individual methods for each APDU call, with appropriate naming conventions to disambiguate between APDU and future protocols.

[1] See Section 2.3.3

3.2.1 Example APDU Methods

The APDU protocol provides several calls for updating files via a binary interface. These are `APDU_READ_BINARY`, `APDU_WRITE_BINARY`, `APDU_UPDATE_BINARY` and `APDU_ERASE_BINARY`. Each of these corresponds to a method call of the same name.

APDUs are implemented as a message/response system between the computer and the card, but I wanted to hide this from the applications as much as possible. Therefore, each method also has the return values expected from the card in the method signature, so the application programmer only needs to make one library call do request the data.

The interfaces provided can be seen in the library header file, an excerpt of which is given in Figure 3.2

```
int apdu_read_binary(card_ref ref,
                     int offset, int length,
                     char* returneddata, int* returnedlength);

int apdu_write_binary(card_ref ref,
                      int offset, int length,
                      char* data);

int apdu_update_binary(card_ref ref,
                       int offset, int length,
                       char* data);

int apdu_erase_binary(card_ref ref,
                      int offset, int length);
```

Figure 3.2: Example Library Interfaces

3.2.2 Implementation of Library Functions

The library functions are merely wrappers to send requests to the daemon process which actually relays them to the card. They are therefore merely a serialisation of the parameters into the protocol layer structures (see Section 3.4). The function then sends the structures to the daemon via the `send_ccid_request()` method in the protocol layer, and is returned the response from the card after the daemon has performed the query. The

function then copies the data from the response to the parameters as appropriate. These can all be seen in Appendix B, Section 162.

3.2.3 Library Documentation

As a library for use by other application programmers, documentation of the API is essential. This is one of the reasons I decide to write the project in CWEB. CWEB introduces the idea that documentation should be written simultaneously with code, and hence all the documentation of the interfaces is included in the CWEB output.[2] Figure 3.3 shows a page of CWEB output, along with the source which generates it in Figure 3.4.

I will also convert the relevant sections of the documentation into UNIX `man` Manual pages for ease of access to the API and function lists.

3.2.4 Selecting a Card

Figure 3.3 also shows the documentation for how to select which card the application wants to communicate with. The computer which the application is connecting to may have several smart-card readers (or a single reader with multiple slots). Since my project was specifically written to cope with this situation, there needs to be some way of selecting cards to be used.

This is accomplished via two mechanisms. Firstly connection to the next available card can be requested, which may block until one is available or may return immediately with a failure if none is availabl.. Secondly, filter can be specified to be applied to a specific APDU file (see Section 2.3.3) to select a card which has that feature. This is the same mechanism mentioned for configuration files in Section 4.5.1.

Exactly how this works is given in the documentation for the `get_card_reference()` function which can be seen in Appendix B, Section 88. For convenience I have reproduced this below.

3.3 Daemon

This is the section of the driver which handles direct communication with the USB devices, and controls access from the client applications. There are two main parts to this. Firstly there is a lot of general overhead of writing a UNIX resident process and doing socket IO to the clients. The second main section was serialisation of commands for the USB communication. I

[2]See more in Appendix B, and the full documentation, details of which are given in the appendix.

102. Function *apdu_read_records( ).* This will read records from an EF. If *ef* is set to *CURRENT_EF* then records will be read from the currently selected EF. Otherwise they will be read from the specified EF. The record to start reading from are specified by *type* and *record*. If *type* $\in$ {*FIRST*,*LAST*,*NEXT*,*PREVIOUS*} then the appropriate record will be read from. If *type* $\equiv$ *SPECIFY*, then the record number will be read from *record*. *length* bytes of data are read from the starting record.

You must *malloc(*2*) length* **bytes of** *returneddata*. If no data is returned, then *returnedlength* will be `0`.

The return value from this function will be the APDU status that the card returns.

int *apdu_read_records (***card_ref** *ref,* **int** *ef,* `RECORD_TYPE` *type,* **int** *record,*
int *length,* **int** **returnedlength,* **char** **returneddata);*

88. Function *get_card_reference( ).* This function allows you to request a connection to a smart-card. You can specify the card to request in several different ways, some of which are blocking, and some are non-blocking.

Blocking requests:
A request for the next available smart-card can be made. This request will block until either a smart-card is inserted, or one in use by a different card is made available.

Non-Blocking requests:
A request can also be made which will return a card if there is one available, but will return immediately with a `NOT_AVAIL` message if there are no free cards.

Specifying cards
Smart-cards can be specified more precisely by giving a pattern to match against the contents of the card. If a file and contents are specified, then any candidate card to be returned will be checked to see if that file exists, and if the contents match the string given in the second parameter. If *file* $\equiv \Lambda$, then no checks will be performed. Otherwise, if *content* $\equiv \Lambda$ then the file will be checked for existence, but not for content.

Possible Requests

Request	Blocking	Description
`BLOCK_NEXT_AVAIL`	yes	Blocks until a card is available and returns a descriptor.
`AVAIL`	no	Returns the error `NOT_AVAIL` if a card isn't available. This will not block

The method has the signature:
int *get_card_reference*(*card_ref** ref, `CARD_REQUEST` request, **char*** file, **char*** content);

Figure 3.3: Example Output from CWEB

```
@* Function |apdu_read_records()|.
This will read records from an EF. If |ef| is set to |CURRENT_EF| then
records will be read from the currently selected EF. Otherwise they will
be read from the specified EF. The record to start reading from are
specified by |type| and |record|.
If |type| $\in \{$|FIRST|,|LAST|,|NEXT|,|PREVIOUS|$\}$
then the appropriate record will be read from. If |type == SPECIFY|, then
the record number will be read from |record|. |length| bytes of data are
read from the starting record.

{\bf You must |malloc(2)| |length| bytes of |returneddata|}.
If no data is returned, then |returnedlength| will be |0|.

The return value from this function will be the APDU status that the
card returns.
@<Exported Library Functions@>=
int apdu_read_records(card_ref ref, int ef, RECORD_TYPE type,
            int record, int length, int* returnedlength, char* returndata);

@* Function |get_card_reference()|.

This function allows you to request a connection to a smart-card.
You can specify the card to request in several different ways, some
of which are blocking, and some are non-blocking.

{\bf Blocking requests:}

...

{\bf Possible Requests}

\halign{\hfil \it # & # & # \hfil \cr
\bf Request & \bf Blocking & \bf Description \cr
|BLOCK_NEXT_AVAIL| & yes & Blocks until a card is available
                           and returns a descriptor. \cr
|AVAIL| & no & Returns the error |NOT_AVAIL| if a card isn't
               available. This will not block \cr
}
```

Figure 3.4: CWEB Source Code for Figure 3.3

decided that for manageability this should be split into a separate file of source code.

3.3.1 UNIX Process Management

A daemon process in UNIX such as I needed here must detach itself from the console and run in the background while other tasks are performed. This is performed by doing a `fork()` to create a separate process, and then causing the parent process te exit, while leaving the child running. The code for this is given in Appendix B Section 34, with detailed documentation.

Having left the console, there is no longer an error stream to print output to. For logging I decided to use the system logger which collects logs from most of the daemons on the system. This was accomplished using the syslog C library, and my wrappers for configuring the syslog output are in Appendix B Section 136.

Finally, if there is no console we do not have a convenient way to send signals or control messages to the process. Also mentioned in Section 3.3.2, you can use a second copy of the daemon to communicate with the first one. If there is already an instance running, then there is a second socket open which can cause the existing copy to perform actions such as re-scanning the USB bus, or exiting the program.

3.3.2 Socket Communication

There are two ways to implement socket communication in C. One is by using a separate thread per open socket. This is at first sight a simple option, however, this brings the complexity of then managing concurrent accesses to resources and implementing some sort of locking to control this. Therefore, I decided to use a single threaded design which avoids these problems, and is also theoretically more efficient. This is done using the `select()` call in C. This also allows the use of blocking asynchronous IO, rather than polling all the sockets for data. Select takes a list of all the currently open sockets, and returns a list of those which have data waiting and need to be read from. Most of the time in the program it is therefore spent in a single loop, blocked in the select call. Because the code is blocked on IO, rather than sitting in a tight loop and polling, the Operating System scheduler does not need to continually allocate it time while it is waiting for data, which is a lot more efficient. This can be seen in Appendix B Section 20.

There are several sockets in use, of several types. Firstly, there are two named sockets. These are implemented as files with special properties, and allow anyone to communicate with the process bound to that socket. The

main socket which client applications write to is world writable, so that any application can connect to it and present credentials to try and get access to a card. The second one is a control channel by which the super-user can send messages to the running application. This is designed to be used by system such as hotplug[4] to notify the system when new USB devices have been added.

When applications request a connection on the named socket, they are allocated a connection-specific anonymous socket which only they can access. This corresponds to a single session with a single smart-card. Applications wanting to access more than one card can open multiple connections. These sockets are all then added to the list which is passed to `select()`. When notified of data on a socket, the daemon services the request for that card, and returns the result over the socket to the client.

Socket Security

Socket communication in implemented in Linux by writing to an area of memory which the process was given access to by the Operating System when it opened the socket. Any Operating System which has memory protection can therefore ensure that a socket cannot be written to except by that process, since other applications cannot access its address space. If this is assumed to be the case, then once a socket has been opened to a particular application it is guaranteed to be the same application which is writing to that socket. Therefore, possession of a socket file descriptor can be used as an authentication token. This relies on the assumptions that the program itself won't give access to that memory location – but if the application is subverted in that way then it can perform malicious accesses itself, and such protection is outside the scope of this project – and the Operating System security. The Operating System (particularly in the case of Linux, but also in general) enforces the security of sockets using the memory protection mechanisms. It can be seen easily that if a malicious process can break the memory protection system, then they can access the USB device directly, or the memory of the client application, and does not need to attack the open socket. Therefore, replying on sockets as authentication tokens is acceptable.

3.3.3 Managing USB devices

When the daemon initially starts it does a scan of the USB bus to find any devices that are currently attached. This scan can be re-run at any time via messages sent over the control socket, and such a message would be sent by

a system like hotplug if it discovered a CCID device being attached. The scan preserves any existing devices and is therefore safe to run at any point. The latest development version of LibUSB allows you to scan all the USB busses for devices matching a particular pattern in their USB class, type and vendor IDs. I scan for devices matching the USB CCID Class ID (`0x0B`), which is defined in the CCID Class-spec[7].

Once a card reader has been detected an entry is put into the list of devices for each card slot the reader has. From that point on, each slot is treated as a separate device, and the slots can be 'owned' by different programs simultaneously. All the slots are then checked to see if they contain a smart-card, and if so added to the list of devices accessible by applications.

3.3.4 USB Protocols

Once the request for a command get to the daemon process then it needs to be encoded to be sent to the smart-card. There are two layers to this, First an APDU Message block must be built up. This is defined in the ISO 7816[2] standard and described in the next section Then this block has to be embedded in a CCID command message block, as described below. The CCID protocol is the USB wire protocol, which sends a command to the card reader. The reader then strips off the CCID layer and sends the APDU command to the card itself. Return messaged from the card are encoded in a similar two-level scheme to be sent back to the daemon.

APDU Protocol

When a request (encoded using the protocol in Section 3.4) has been received by the daemon, it checks the protocol flag on the data structure. This is used to switch which method is called to decode it. For APDU requests it is passed to the `decode_apdu_req` function, which can be seen in Appendix B, Section 81.

A valid APDU starts with a header of 4 bytes. The first two specify the command class and specific instruction, and the second two are parameter bytes. This is optionally followed by an arbitrary length of data attached to the command (with its length) and the expected length of any reply. This is illustrated in Figure 3.5.

The decode function takes a structure as read off the network and converts it into the byte stream to be sent over the USB. To do this, we switch on the APDU type, and call another block of code. This sets the class and instruction header bytes and copies the fixed-length parameters into the appropriate parts of the parameter bytes, as specified in the ISO standard.

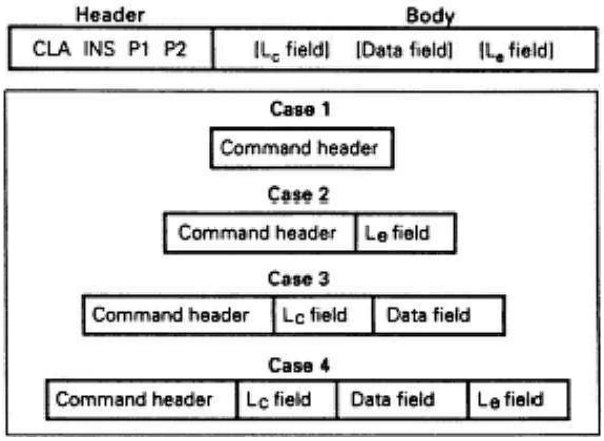

Figure 3.5: APDU Command Structure (taken from ISO 7816 [2])

Attached data is stored in the same part of the structure for all protocols, and can be copied onto the end of the data block in all cases. If the particular command expects data back from the command then it also sets the return length byte. This byte stream is then passed to the CCID layer.

When a reply is received from the card it is also encoded as an APDU block. The reply contains first an optional, variable length data block of the length specified in the command that caused the reply. Secondly, it contains two mandatory status bytes which indicate the success or otherwise of the command. The APDU response structure which is passed over the network correspondingly contains a status word, and the generic response structure contains a pointer to a byte stream. These are copied from the byte stream received from the network and the structure is returned to be sent to the client application.

CCID Protocol

The decoded APDU requests need to be wrapped in a CCID block to be sent to the card reader. This is a USB-level protocol and deals with the `T=0` / `T=1` distinction. A CCID header contains the type of CCID command (`PC_TO_RDR_TOAPDU`), the slot in the device to send the command to, and the length of the APDU command. The resulting byte stream (10 bytes of CCID header, followed by 4 bytes of APDU header, followed by 4 bytes of send and receive length, plus the data to send with the command) is written to the USB device using a bulk out command to LibUSB.

When receiving a reply from the USB device, the CCID header is validated and then stripped off.

3.4 Library-Daemon Protocol

The protocol used for communicating over the sockets between the client applications and the daemon is in essence a serialisation of the APDU or other command which is to be sent to the card. This serialisation is implemented by having a set of fixed-size parameters stored in a structure, which is written to the network byte-wise, and an optional arbitrary-length block of data, the length of which is specified in the structure. The receiver can therefore read just the structure, then find out how many more bytes to expect on the socket.

This is implemented using a series of `struct`s which have been overlapped using the `union` operator. Unions enable the specification of a set of mutually exclusive options which are stored in the same space. Depending on context, the same memory location is accessed as a different type. With this technique, all the different types of APDU have different numbers of parameters, but the same type can be used whilst only needing to store the largest set at use at once.

The full structures can be seen in Appendix B, Section 146, but I have included a sample in Figure 3.6. In each case of a union there is a variable which indicates which member of the union is to be used to access the data and indicates the types in use. Everything within the protocol-specific structures is a fixed size, and there is a protocol-agnostic method of appending extra data.

All the values in the structures are fixed length variables, rather than reference types, so that when reading the structure over the network a fixed number of bytes must be read. The exception to this is the pointer to the variable length data. This is defined at the top level so that all protocols have access to a variable length data string, but the network code can be the same for all. When writing to the network you must send the structure, followed by exactly `dsize` bytes, which are then copied into a memory location at the other end and assigned to `data`, or `data` set to `NULL` iff `dsize == 0`.

There is a similar structure for returning data from the card organised in the same way, with the only APDU return value being two status bytes in all cases, and optionally a block of data returned from the card. This is handled in the same way as the request structure described above.

```
struct ccid_request_struct {
   PROTOCOL_TYPE protocol;
   union {
      struct apdu_request_struct apdu;
      ...
   };
   int dsize;
   char* data;
};

struct apdu_request_struct {
   int apdu_type;
   union {
      struct apdu_binary binary;
      struct apdu_record record;
      ...
   } ;
};

struct apdu_binary {
   MODE mode;
   int offset;
   int length;
};
```

Figure 3.6: Structures for Storing APDU Commands

Chapter 4

Evaluation

The success of this project is evaluated on several basis. Firstly, the project set out to provide an interface for all of the services provided by an ISO 7816 Smart-Card. Secondly, the interface presented to programmers using the library should be as simple and easy to use as possible. Finally, the implementation should be well written, stable and easy to maintain, and secure.

4.1 Serviceability

As I stated at the beginning, there was a lot that I could have implemented in this project. There are two CCID-card protocols, and three application-level protocols. All the protocols are also extensible with proprietary commands. I started implementing only the T=0 and APDU protocols respectively, but writing the protocols and programs such that it would be easy to add the others in later.

There are three areas that the code must be extensible to support the rest of the protocols: the library API, the application-daemon protocol and the USB interfacing code.

4.1.1 Library API

Since this has been provided simply with individual method calls for each APDU function, it is trivial to extend in a backwards-compatible way by adding more functions. The library uses a prefix of the protocol on all the method names. Consequently, you have a separate name-space for each protocol's functions, and new protocols can be added without worrying about overlapping command names.

4.1.2 Application-Daemon Protocol

The protocol used to communicate over the sockets between the two sections of the project obviously needs to be extensible to handle the new protocols. It has been designed with this in mind. Protocols can be added by putting another option in the top level union, and adding another constant to indicate the protocol in use. This is not necessarily completely backwards-compatible, if the protocol has a larger block of fixed-length data than the current ones. However, since this protocol is only used by the library code and the daemon, which will be distributed together, applications do not use it and there are no compatibility problems.

4.1.3 USB Code

The translation to USB and ISO 7816 protocols is done entirely within the daemon, so there are few, if any, issues with backwards-compatibility. As far as extensibility is concerned, the USB layer passes each method through functions to do the encoding of the commands into the appropriate protocols. Adding support for new protocols is fairly easy as C programs go, however, would have been easier in a language with support for classes, for example Java. In that case it would simply require overriding a method, and the calling code does not need to know that there is a new protocol in use.

4.1.4 T=0 and APDU

Since there was so much work involved in this project, the ISO 7816 standard is very large, and to fully implement a single one of the protocols would have taken more time that I had available for this project. Therefore, I had to stop implementing all of the APDU protocol, and it does not yet support all of the commands. However, I have implemented a sufficient subset to write some test programs to validate the rest of the project and all of the methods which haven't been implemented are present as stubs for later implementation. The USB-level encapsulation is complete for T=0 APDU commands.

4.2 Simplicity

As part of this project I wrote a test application to demonstrate that the code was functioning, and to show the ease of use for application programmers. The code for this can be found in Appendix A. The program merely waits for a Cambridge University ID Card to be inserted, and then reads out the card

number and assorted details. The code in the Appendix has been written along with documentation describing its operation, so I have included just the sequence of library calls necessary in Figure 4.1. As can be seen, this is a very simple interface, allowing programmers access to what they want, whilst hiding most of the complexity.

```
char* carddata;
int length;
...
get_card_reference(&ref, BLOCK_NEXT_AVAIL, "FD03",
"<0-9>{8}<a-z>{2}<0-9>{4}<a-z>0.000");
apdu_read_binary(ref, 0, 20, carddata, &length); release_card(ref);
```

Figure 4.1: Example Application Code

4.3 Stability

Since this was too large a project to complete in its entirety, I chose to build the framework within which all the components sit first. This means that it is currently not possible to test all of the available options, which would be the traditional test regime for this sort of software. Unfortunately, it has meant that many of the functions have not been fully tested, although enough has been implemented to produce very simple test applications, as can be seen in Appendix A. Many parts of the driver are very generic, and don't depend on the protocol data being sent over them and this framework is complete. The socket communications, and the UNIX systems programming parts have all been tested and are stable.

4.4 Security

The security of a complex system is never an absolute, but rather an exercise in risk management. The security goals were given in Chapter 2, and I shall go over them again here to see how well the project matches my original assessment. There are also some security issues that I hadn't considered in the original planning, but which arose later.

The main security policy that I wanted to enforce with the driver, was that any session communicating with a smart-card could only have one application communicating with it. As I said in Section 3.3.2, the operating system memory protection enforces the fact that once opened, a socket can

only be written to by that application. Because I am using sockets as authentication tokens, only the application with a socket is allowed to send commands to to a particular reader slot while it is bound to an application. The setup and tear-down operations on sockets also force power up and power down events on the smart-card, so it is guaranteed to be in the reset state after communications have finished. If the operating system memory protection failed, or the application using my library allowed arbitrary command insertion into the sockets then my security assumptions are false, but that is both outside the scope of the project, and would also result in several other methods of breaking the security.

An issue I hadn't considered was restricting access to certain programs or users. I was not initially concerned with which application could use a card, because the smart-cards have their own security mechanism built in. The main issue was with access to the card once an application had authenticated to the card. However, with the system as it is there is a potential for a denial of service. An application can start communicating with a card, and stop other applications from doing so. It doesn't need any special privileges to do this. A second minor worry is for cards which are authenticating merely by their presence, and don't contain any secrets. This can be handled in a lot of cases via UNIX permissions and Linux Pluggable Authentication Modules (PAM).

The UNIX permissions system allows the restriction of which users can write to the socket. This can be done easily if the restriction is on a per-user level by adding the appropriate users to a UNIXgroup, and only allowing this group to write to the socket. Changing to project to support this would be a relatively simple matter of adding a configuration file option and changing the line of code which sets the permissions.

If the restriction needs to be per-application then this is a lot more difficult. Some solutions exist, but not in the standard Linux kernel. Projects such as SELinux and GRSecurity allow more fine-grained access control.

Pluggable Authentication Modules for Linux can perform actions such as changing the ownership of a sockets when the user logs in, can can perform different actions if the user is physically at the terminal or is a remote user. If the model is that only users physically at the computer will be using smart-cards, and there can be only one such user at a time, then the big stick security principle[1] is completely acceptable.

[1] "Whoever has physical control of the device is allowed to take it over" - Frank Stajano[9]

4.5 Additional Features

There are several features which I had planned to include in the project, but unfortunately did not fit within the available timescale.

4.5.1 Interrupt-based events

Most, if not all, of the modern card readers have support for both bulk IO and interrupt-based communications. One of the features I wanted to offer with this project was support for handling insert/remove notifications via this interrupt mechanism. Currently this has to be implemented by regularly polling the status of each slot in the reader. This is not a particularly elegant way of implementing this, as it means the process has to be using the processor and the USB bus each time it checks, rather than just blocking on a read call.

The plan was to have a similar mechanism to the one defined in Section 3.2.4 to allow applications to bind themselves to a particular card or type of card. This association between the card and the application would be defined in a configuration file for the daemon. When a card is inserted, the daemon would check it against the features listed for cards in the configuration file and if one matched then the daemon would launch the appropriate program to deal with it. A second, similar approach would be to have running applications be sent asynchronous messages when a match for a card is inserted, and allow them to register that while running.

4.6 Testing

Ideally, full testing would be done by emulating the hardware device and sending to it all the possible commands, and having it reply with all the available errors, along with systematically chosen invalid responses. This is the most robust method of testing a device driver, but requires the most effort. Essentially a second implementation of the standard must be made to do the emulation.

Given the timescale and that a complete implementation of even a single protocol was not feasible, so testing all of the commands at the current status of the project is also not possible. Appendix A contains a sample application which reads from a Cambridge University ID Card and prints the result. This can be used to demonstrate multiple client applications talking to the daemon over the sockets, and the commands being translated to the device.

4.7 Summary

The project has not, unfortunately, yet met all of the goals listed in Section 2.2. Those that are missing, however, are just further instances of things which have already been completed. Enough of the project is working to validate the architecture and to demonstrate that the system works in principle. All of the major elements have examples in place and the result is definitely simpler and easier to use than the current alternatives. Given the timescale to which I was working it was not expected to have a fully working product, but I have produced enough to see that one would be viable.

Chapter 5

Conclusions

The aim of this project was to provide an application-level interface to CCID[7] based smart-card readers and ISO 7816[6] compliant smart-cards which was simpler and easier to use than the current drivers for doing so. As set out in the Introduction chapter there is a need for such an interface.

5.1 Achievements

A two-part driver system for talking to ISO 7816[6] smart-cards was designed and implemented. This is comprised of a constantly-resident daemon process which is responsible for communicating directly with the smart-card readers and cards, and a shared library for application programmers to link to which exports all the APDU commands to the application. Applications communicate with the daemon over UNIX sockets.

To demonstrate this I have written an application to read the information from the Cambridge University ID card. The simplicity of this application shows how easy it is to program using this driver.

5.2 Future Work

The common theme throughout the Evaluation chapter is that the project was considerably larger than either I or my supervisor had anticipated. The amount of work involved in creating a functioning driver was a lot greater than the time I had available. Were I to plan this same project again, I would have started off getting a solid USB CCID and APDU layer working before starting on the Application Interface. However, merely creating a driver was not the main aim of the project, it was to create a good, easy and feature-full interface for other programmers. Given the same time, it

would have been put to better use building on top of an existing interface to the hardware to create a good programming library.

Bibliography

[1] Ccid smart-card library. http://www.matthew.ath.cx/publications/.

[2] *ISO Standard 7816 Section 4.*

[3] libusb project. http://libusb.sourceforge.net/.

[4] Linux hotplugging. http://linux-hotplug.sourceforge.net/.

[5] The universal serial bus. http://www.usb.org/.

[6] *ISO Standard 7816 / BSI Standard 27 816.* BSI, 389 Chiswick High Road, London, W4 4AL, UK, 1987 / 1991.

[7] *Universal Serial Bus Device Class Specification for USB Chip/Smart Card Interface Devices*, first edition, March 2001. http://www.usb.org/.

[8] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation.* Addison-Wesley, 3.6 edition, 1993.

[9] Frank Stajano. *Security For Ubiquitous Computing.* John Wiley & Sons Ltd, Chichester, West Sussex, 2002.

Appendix A

Test Application Code Listing.

Appendix A contains the code for a test application using the project to access the smart card. It will read the card data from a Cambridge University ID Card.

1. Header file includes.

⟨ Include our own header files 2 ⟩
⟨ Include the CCID Card library 3 ⟩

2. Our own header file for this program. This gives us debug output conditional on the DEBUG preprocessor variable and the assert statement to check values are correct.

⟨ Include our own header files 2 ⟩ ≡
```
#include "../share/debug.h"
#include "../share/log.h"
```
This code is used in section 1.

3. The CCID Library headers. This allows us to talk to the cards via a daemon which is managing access

⟨ Include the CCID Card library 3 ⟩ ≡
```
#include "../lib/library.h"
```
This code is used in section 1.

4. The main part of the program.

```
int main(int argc, char **argv)
{
  ⟨ Tester local variables 6 ⟩
  ⟨ Connect to the daemon and request a reference to a card 5 ⟩;
  ⟨ Send a command to the card 7 ⟩;
  ⟨ Exit the program 9 ⟩;
}
```

5. Connecting to the card. We make a library call to setup the connection to the daemon and request access to a card. It returns us a reference to the card we can use to send commands to it.

To see if this is a university card we check for the existence of file "FD03" and its contents

⟨ Connect to the daemon and request a reference to a card 5 ⟩ ≡
```
debug("Getting␣card␣reference\n");
assert(get_card_reference(&ref, BLOCK_NEXT_AVAIL, "FD03",
    "<0-9>{8}<a-z>{2}<0-9>{4}<a-z>0.000") ≥ 0, "Could␣not␣connect");
```
This code is used in section 4.

6.

⟨ Tester local variables 6 ⟩ ≡
```
card_ref ref;      /* a reference to the card we are accessing */
```
See also section 8.

This code is used in section 4.

7. Sending a command. Read the card ID from the card.

⟨ Send a command to the card 7 ⟩ ≡
```
debug("Sending␣command\n");
apdu_read_binary(ref, 0, 20, carddata, &length);
carddata[length] = '\0';      /* make sure it is null terminated */
printf("The␣Card␣ID:␣%s", carddata);      /* print it out. Its mostly ascii */
```
This code is used in section 4.

8.

⟨ Tester local variables 6 ⟩ +≡
```
char *carddata = malloc(21);
int length;
```

9. Exitting the program. After connecting and sending a command, we exit the program.

⟨ Exit the program 9 ⟩ ≡
```
release_card(ref);
free(carddata);
debug("Exitting");
return 0;
```
This code is used in section 4.

Appendix B

Project Code Examples.

Appendix B contains excerpts of code from the project. These excerpts are individual sections which are referred to in the body of the report, and have been taken from the document produced using Knuth and Levy's CWEB system of structured documentation to apply markup in TEX.

The complete listing of source for the project is included on the attached CD, or can be downloaded from *http://www.matthew.ath.cx/publications/*.

20. Procedure *listen_sockets*( ). Loop, running *select*(2) over the sockets and handling the results from them.

```
void listen_sockets( )
{
  ⟨ listen_sockets Local variables 22 ⟩;
  for ( ; ; ) {      /* Loop forever here unless told to exit by a control message */
    ⟨ Add all the active sockets to an fd_set 23 ⟩;
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;
    assert((rc = select(scount, &socks, Λ, Λ, &timeout)) ≥ 0, "Select Error");
      /* check for things to read */
    debug("Connections on %d sockets.", rc);
    debug("checking usb devices for interrupts");
    check_usb_interrupts( );
    if (0 ≡ rc) continue;      /* nothing to read */
    {      /* read from the remaining sockets. */
      if (FD_ISSET(master_socket, &socks))      /* the master socket */
      {
        ⟨ Accept a new connection 24 ⟩;
      }
      if (FD_ISSET(control_socket, &socks))      /* the control socket */
      {
        if (handle_control_message( )) return;      /* handle control message and exit if told to */
      }
      current = client_root;
      while (Λ ≠ current)      /* the client sockets */
      {
        if (FD_ISSET(current→socket, &socks)) {
          if (¬service_client(current)) {      /* connection should be closed */
            slog("Client connection (pid: %d) closed", current→pid);
            close(current→socket);      /* close the connection */
            client_root = removeclient(client_root, current);      /* remove the client from the list */
            free(current);
            current = Λ;
          }
          else current = current→next;      /* go onto the next client */
        }
        else current = current→next;      /* go onto the next client */
      }
    }
  }
}
```

34. Fork from console. We have to *fork*(2) a child proccess and the exit the parent to return to the console, then *setsid*(2) to disassociate ourselves from the tty we were spawned on. Finally, we *chdir*(2) to the root directory so that filesystems can be unmounted.

```
⟨ Fork from console 34 ⟩ ≡
  debug("Leaving␣console....");
  pid = fork( );
  assert(pid ≥ 0, "Error␣in␣fork");
  if (0 < pid) exit(0);
  setsid( );
  chdir("/");
  umask(0);
```

This code is used in section 32.

81. Function *decode_apdu_req*( ). Turns a *ccid_request* structure into **void** ∗ buffer for writing to a USB device. The function returns false if an error occurred.

NOTE: may *malloc*(2) *buffer*. If the function returns *true* you **MUST** use *free*(2) to deallocate it when you are finished.

buffer A pointer to a **void** ∗ buffer

req The structure to decode to the buffer

```
⟨ USB Internal Functions 68 ⟩ +≡
  bool decode_apdu_req(ccid_request * req, void **buffer)
  {
    char header[4];      /* [0] = CLA, [1] = INS, [2] = P1, [3] = P2 */
    header[0] = #00;
    header[1] = #00;
    switch (req→apdu.apdu_type) {
    case APDU_AUTHENTICATE: ⟨ APDU authenticate decode 0 ⟩;
    case APDU_BINARY: ⟨ APDU binary decode 82 ⟩;
    case APDU_CHALLENGE: ⟨ APDU challenge decode 0 ⟩;
    case APDU_CHANNEL: ⟨ APDU channel decode 0 ⟩;
    case APDU_DATA: ⟨ APDU data decode 0 ⟩;
    case APDU_RECORD: ⟨ APDU record decode 0 ⟩;
    case APDU_SELECT: ⟨ APDU select decode 0 ⟩;
    case APDU_VERIFY: ⟨ APDU verify decode 0 ⟩;
    default: return false;
    }
    return true;
  }
```

82. Decoding an APDU Binary command.

```
⟨ APDU binary decode 82 ⟩ ≡
  switch (req→apdu.binary.mode) {
  case READ:
    if (0 ≡ header[1]) header[1] = #B0;      /* set the INS to READ BINARY */
  case ERASE:
    if (0 ≡ header[1]) header[1] = #0E;      /* set the INS to ERASE BINARY */
    *buffer = malloc(3 + 4);      /* payload + header size */
    ((char *) *buffer)[6] = (req→apdu.binary.length & #FF);
    ((char *) *buffer)[5] = ((req→apdu.binary.length & #FF00) ≫ 4);
    ((char *) *buffer)[4] = 0;      /* Lc = null, Data = null, Le = req→apdu.binary.length */
      /* buffer[0] = 0 =¿ extended format Le */
    break;
  case WRITE:
    if (0 ≡ header[1]) header[1] = #D0;      /* set the INS to WRITE BINARY */
  case UPDATE:
    if (0 ≡ header[1]) header[1] = #D6;      /* set the INS to UPDATE BINARY */
    *buffer = malloc((req→dsize & #FFFF) + 3 + 4);      /* payload + payloadsize + header */
    ((char *) *buffer)[6] = (req→dsize & #FF);
    ((char *) *buffer)[5] = ((req→dsize & #FF00) ≫ 4);
    ((char *) *buffer)[4] = 0;      /* Lc = req→dsize, Data = req→data, Le = null */
      /* buffer[0] = 0 ⇒ extended format Lc */
    void *b = *buffer + 7;      /* WTF?! */
    memcpy(b, *buffer, req→dsize);
    break;
  case APPEND: default: return false;
  }
  header[3] = (req→apdu.binary.offset & #FF);
  header[2] = ((req→apdu.binary.offset & #7F00) ≫ 4);
  memcpy(*buffer, header, 4);
  break;
```

This code is used in section 81.

88. Function *get_card_reference*( ).

This function allows you to request a connection to a smart card. You can specify the card to request in several different ways, some of which are blocking, and some are non-blocking.

Blocking requests:

You can make a request for the next available smart card. This request will block until either a smart card is inserted, or one in use by a different card is made available.

Non-Blocking requests:

You can make a request which will return a card if there is one available, but will return immediately with a `NOT_AVAIL` message if there are no free cards.

Specfiying cards

Smart cards can be specified more precisely by giving a pattern to match describing them. If you specify a file and contents, then any candidate card to be returned will be checked to see if that file exists, and if the contents match the string given in the second parameter. If $file \equiv \Lambda$, then no checks will be performed. Otherwise, if $content \equiv \Lambda$ then the file will be checked for existance, but not for content.

Possible Requests

Request	Blocking	Description
`BLOCK_NEXT_AVAIL`	yes	Blocks until a card is available and returns a descriptor.
`AVAIL`	no	Returns the error `NOT_AVAIL` if a card isn't available. This will not block

⟨ Exported Library Functions 88 ⟩ ≡

```
int get_card_reference(card_ref * ref, CARD_REQUEST request, char *file, char *content);
```

See also sections 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 119, 121, 123, 125, 127, and 129.

This code is used in section 85.

89. `CARD_REQUEST` Type definition. This type selects what type of request to make.

⟨ Library Type Definitions 89 ⟩ ≡

```
typedef enum {
  AVAIL, BLOCK_NEXT_AVAIL
} CARD_REQUEST;
```

See also section 90.

This code is used in section 85.

90. Card references. A card reference contains the information about how to connect to one smart card. A card reference must be retrieved from a *get_card_reference*( ) call, and should be relinquished with a *release_card*( ) call when finished with.

Since sockets are being used to identify cards in the cardd, this is merely a socket identifier and info block.

⟨ Library Type Definitions 89 ⟩ +≡

```
struct card_ref_struct {
  struct sockinfo_struct {
    sa_family_t family;
    char sockpath[MAXSOCKETPATH];
  } info;
  int socket;
};
typedef struct card_ref_struct card_ref;
```

91. Implementation of function.

```
int get_card_reference(card_ref *ref, CARD_REQUEST request, char *file, char *content)
{
  socklen_t len;
  pid_t pid;
  int rc;
  debug("socket: %d", ref→socket);
  ref→socket = socket(AF_UNIX, SOCK_STREAM, 0);
  debug("getting socket, errno: (%d) %s", errno, strerror(errno));
  assert(ref→socket > 0, "Can't get socket fd");
  debug("socket: %d", ref→socket);
  ref→info.family = AF_UNIX;
  strcpy(ref→info.sockpath, SOCKETDIR);
  strcat(ref→info.sockpath, "/master");
  debug("master socket: %d, %s", ref→socket, ref→info.sockpath);
  len = sizeof (ref→info.family) + strlen(ref→info.sockpath) + 1;      /* connect */
  debug("connecting to daemon");
  if ((rc = connect(ref→socket, (struct sockaddr *) &(ref→info), len)) < 0) return rc;
      /* send our pid */
  pid = getpid( );
  debug("sending pid (%d)", pid);
  if ((rc = send(ref→socket, &pid, sizeof (pid_t), 0)) < 0) return rc;
  return 0;
}
```

136. Logging functions.
This file contains functions for sending log messages to syslog, and to perform conditional logging.

137. Header files and global variables. We need syslog header files, and also debugging code. We store a static global variable which governs whether to use syslog or to print messages on *stderr*.

```
#include <syslog.h>
#include <stdarg.h>
#include <errno.h>
#include "log.h"
#include "debug.h"
  bool _do_syslog = false;
```

138. Exported interfaces. The file `log.h` exports several macros and functions to other files.

```
⟨ log.h 138 ⟩ ≡
#ifndef __CCID_LOG_H
#define __CCID_LOG_H
#include "types.h"
#ifdef DEBUG
#define assert(a, b) do
  {
    bool __T = a;
    fprintf(stderr, "[Assert] %s\n", __T ? "true" : "false");
    if (¬__T) real_assert(b);
  }
  while (0)
#else
#define assert(a, b) if (¬(a)) real_assert(b)
#endif
  void real_assert(char *message);
  void slog(char *fmt, ...);
  void setup_syslog(bool log);
#endif
```

139. Procedure *real_assert*( ).

```
  void real_assert(char *message)
  {
    if (0 ≠ errno) {
      char *errstr = strerror(errno);
      slog("[Assert] failed with errno %d/%s: %s\n", errno, errstr, message);
    }
    else slog("[Assert] failed: %s\n", message);
#ifdef DEBUG
    exit(1);
#endif
  }
```

140. Procedure *slog*( ).

```
void slog(char *fmt, ...)
{
  va_list args;
  va_start(args, fmt);
  if (_do_syslog) vsyslog(LOG_NOTICE, fmt, args);
  else {
    fprintf(stderr, "[ccid-cardd] ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
  }
  va_end(args);
}
```

141. Procedure *setup_syslog*( ).

```
void setup_syslog(bool log)
{
  _do_syslog = log;
  if (log) openlog("ccid-cardd", 0, LOG_DAEMON);
}
```

146. CCID structures. These structures are a generic wrapper which may contain one of several protocols.

```
⟨ Definitions of CCID structures 146 ⟩ ≡
  struct ccid_request_struct {
    PROTOCOL_TYPE protocol;
    union {
      struct apdu_request_struct apdu;
    };
    int dsize;
    char *data;
  };
  struct ccid_response_struct {
    PROTOCOL_TYPE protocol;
    union {
      struct apdu_response_struct apdu;
    };
    int rsize;
    char *rdata;
  };
  typedef struct ccid_request_struct ccid_request;
  typedef struct ccid_response_struct ccid_response;
```

This code is used in section 144.

147. APDU BINARY structure. Encodes APDU BINARY commands

```
⟨ Definitions of APDU structures 147 ⟩ ≡
  struct apdu_binary {
    MODE mode;
    int offset;
    int length;
  };
```

See also sections 148, 149, 150, 151, 152, 153, 154, 155, and 156.

This code is used in section 144.

148. APDU RECORD structure. Encodes APDU RECORD commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_record {
    MODE mode;
    int ef;
    RECORD_TYPE type;
    int record;
    int length;
  };
```

149. APDU DATA structure. Encodes APDU DATA commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_data {
    MODE mode;
    TAG_TYPE tagtype;
    int tag;
    int length;
  };
```

150. APDU SELECT structure. Encodes APDU SELECT commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_select {
    SELECT_MODE mode;
    int rtemplate;
    int rsize;
    union {
      struct {
        FILE_TYPE type;
        int DF;
      };
      struct {
        bool relative;
      };
    };
  };
```

151. APDU VERIFY structure. Encodes APDU VERIFY commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_verify {
    bool global;
    int reference;
  };
```

152. APDU AUTHENTICATE structure. Encodes APDU AUTHENTICATE commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_authenticate {
    bool internal;
    bool globalsecret;
    int algorithmid;
    int secretid;
    int responselength;
  };
```

153. APDU CHALLENGE structure. Encodes APDU CHALLENGE commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_challenge {
    int maxlength;
  };
```

154. APDU CHANNEL structure. Encodes APDU CHANNEL commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_channel {
    bool open;
    int channelno;
  };
```

155. APDU REQUEST structure. This is a union of the various APDU structures for the different commands. the *apdu_type* field defines which member of the union should be accessed.

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_request_struct {
    int apdu_type;
    union {
      struct apdu_binary binary;
      struct apdu_record record;
      struct apdu_data data;
      struct apdu_select select;
      struct apdu_verify verify;
      struct apdu_authenticate auth;
      struct apdu_challenge challenge;
      struct apdu_channel channel;
    };
  };
```

156. APDU RESPONSE structure. This encodes responses to an APDU command.

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_response_struct {
    APDU_TYPE apdu_type;
    int status;
  };
```

157. Functions which send and/or receive CCID structures.

```
⟨ Declarations of CCID functions 157 ⟩ ≡
  int send_ccid_request(int skt, ccid_request req, ccid_response *res);
  int get_ccid_request(int skt, ccid_request *req);
  int send_ccid_response(int skt, ccid_response res);
```

This code is used in section 144.

162. Function *send_ccid_request*( ). **This may** *malloc*(2) *req*→*rdata*. **If the return value is** 0 **and** *req*→*rdata* ≠ Λ **then you must** *free*(2).

```
int get_ccid_request(int skt, ccid_request *req)
{
  int ret = 0;
  debug("get structure");
  if ((ret = recv_bytes(skt, (void *) req, sizeof(ccid_request))) < 0) return ret;
      /* get structure */
  print_ccid_request(req);
  ret = 0;
  debug("get data (size: %d)", req→dsize);      /* get data */
  if (0 ≡ req→dsize) {
    req→data = Λ;
    return 0;
  }
  else {
    req→data = malloc(req→dsize);
    if ((ret = recv_bytes(skt, req→data, req→dsize)) < 0) {
      debug("Freeing...");
      free(req→data);
      req→data = Λ;
      return ret;
    }
    debug("Got data");
  }
  return 0;
}
```

Part II Personal Project Proposal

CCID Smart-card library

Matthew Johnson
Trinity Hall
mjj29@cam.ac.uk

May 6, 2004

Project Originator: Dr. M. Kuhn

Project Supervisor: S. J. Murdoch

Signature:

Director of Studies: Dr. S. Moore

Signature:

Project Overseers: Dr. J. Bacon and Dr. J. Daugman

Project Proposal

Background

The idea (kindly suggested by Dr Kuhn) is to develop a driver system for GNU/Linux for accessing USB-based smart-cards and readers, which can handle multiple devices and cards and multiple registered applications using those cards.

The cards are ISO-7816 smart-cards, a subset of which are used in both University cards and EMV[1]-compliant credit cards, the readers are generic CCID[2]-compliant USB smart-card readers.

The current situation with Linux smart-card support is a series of disjoint drivers from the Musclecard project, mainly designed for serial or parallel connected devices. They also do not support having multiple smart-cards talking to multiple separate applications. It is to be expected that with the CCID standard (previous smart-cards used mostly custom protocols) and the ease of use of USB, these devices will rapidly dominate the marketplace.

There are several parallels between smart-cards and USB devices, particularly with how they are managed on insertion. There is a program for Linux called hotplug, which manages USB devices. This has a daemon which runs in the background, and receives notification from the kernel about USB events, such and insertion of a new device. Hotplug gets from the device a USB class, a vendor ID and a product ID. The daemon checks a database of those ID strings, and adds the appropriate modules to the kernel to manage the device, and also can launch programs which have been registered as handling a certain class of device.

What is needed for smart-cards is a generic library for accessing smart-cards, which will also perform the same job as hotplug does for USB devices. More formally, it will manage the permissions on smart-card reader devices such that multiple applications can securely access their cards. Applications can also be set as handlers to be launched for certain types of cards or can be run themselves to listen for the next card inserted. This is similar to the hotplug program for USB devices, however, there's is not an obvious architecture in place for classifying cards as with USB devices.

[1] Europay-Mastercard-Visa: *http://www.emvco.com*

[2] Chip/Smart-Card Interface Devices - specifications on
http://www.usb.org/developers/devclass_docs/ccid_classspec_1_00a.pdf

Project Aim

The aim of the project itself will be to produce a driver, split into two sections, which can be used by applications wanting to access smart-cards. Some trivial test programs will be used to demonstrate this. The two sections of the driver would be a daemon process, similar to hotplug - and launched by hotplug when a smart-card reader was connected. The second part is a library which would be used by application software to provide an API for accessing the smart-card, and for communicating with the daemon.

The daemon part of the driver would be launched by hotplug when a CCID-compliant reader is connected. This would then wait for insertion of a card and depending on various settings it would hand over control of the card to some application. This could either be a one off application notifying the driver that it is waiting for the insertion of the card, or pre-registered as the application which handles that type of card.

The other section would be a library used to access the cards. This would be used by any application wanting to use the driver, and would communicate with both the daemon process and directly to the card when control was handed over. Since the CCID specification specifies several levels of access to the card, the API provided by the library functions would have to allow access at any of these levels of abstraction. Obviously an API specification for use by other programmers (in the form of Unix Manual pages) should be provided.

There are several challenges here. Firstly, at a basic level I need to work on by programming in C, since I haven't done much of this. Moving from Pascal and Java shouldn't be too much of a problem, however. Secondly, as I have said, USB provides a nice interface for classifying devices. The CCID spec doesn't provide anything equivalent. There are several possible strategies to look at, such as allowing applications to provide scripts in some language which would decide if a card was usable by that application. There are obvious security issues here which need to be thought through.

The applications that will be used to demonstrate the library will mainly be trivial test applications, however, I shall include a tool which is useful in itself, which is one for performing interactive sessions with the card.

Assessment Criteria

A typical application use for this would be something like a Linux Pluggable Authentication Module for authentication, which would require a card inserted to log in. Success of this project will be determined when the test applications written using several different levels of the API can access cards independently and reliably.

Timetable

Weeks	Work
Michaelmas Term	
1-2	USB & CCID spec reading + C practice
3-4	Design of daemon API, library API, test specifications
5-6	Implementation of daemon
Milestone:	daemon can report details of card on insertion
Christmas Holidays	Implementation of library
Lent Term	
Deliverable:	Progress Report
7-8	Implementation of library
Milestone:	Library functions which can communicate with the daemon and the card
9-10	Testing & collecting scalability data
11-12	Writing Up
13-14	Writing Up
Easter Holidays	(Revision)
Easter Term	
15-16	Writing Up
17-18	Time used for finishing dissertation if necessary
18-19	Time used for finishing dissertation if necessary
Deliverable:	Final dissertation

Resources

I shall be doing most of the development on my personal machines. These comprise of a Dell Intel-based laptop and a recent AMD PC both running Debian GNU/Linux, with the files stored on a Via Epia machine with redundant storage.

Backup Arrangements

Backup arrangements will be made using a selection of machines distributed across several colleges and departments, and the University Archive server. Code will be stored in CVS on one machine, with continuous mirroring on a separate machine, with nightly backups of the CVS repository taken to the other machines distributed across Cambridge.

Special Resources

The Security Group are providing several CCID Smart-Card readers and ISO-7816 smart-cards.

Supervisor

Steven Murdoch of the Security Group has agreed to supervise me, at Dr Kuhn's suggestion. Dr Kuhn will also be providing technical advice.