Individual Project Report

Name : Nicholas Towers (ncet@doc.ic.ac.uk)
Course : MEng Computing IV
Supervisor : Peter Cornwell (pjc@doc.ic.ac.uk)
Second Marker : Stuart Cox (smc@doc.ic.ac.uk)

Title : Implementation of a Generalised API for Application Control by Telephony Devices

June 2000

Abstract

Increasing use of data communications over telephony services is leading to widespread application software development. At the same time, the proliferation of messaging technologies such as SMS[1] has consolidated the major market share of telephony-based communications, making a wholesale migration to personal internet services unlikely in the near future. Support by application programs of multiple services such as DTMF[2] tones, voice control, SMS and WAP[3] therefore becomes attractive to ensure the widest possible customer access.

This report describes the design, implementation and evaluation of an API[4] for application control by generic telephony devices. Several specific solutions already exist for communication between handsets and systems such as telephone banking and customer service centres. However, commercial opportunities for these applications are growing rapidly with the widespread uptake of mobile telephony, in addition to the large installed base of fixed line telephones. No architecture exists to permit re-use of client interface software.

The API developed in this project addresses these problems by offering an interface to telephony (DTMF, voice, SMS, WAP) services which communicates with application programs through a service daemon. Access to the applications via remote control over the Internet (WWW[5], telnet, email) is also provided. The applications developed during the project to evaluate the API demonstrate effective communication with users employing each of these services. Further work undertaken has led to the implementation of a library interface for programmers, enabling effective commercial use of the API.

[1]Short Messaging Service
[2]Dual Tone Multi Frequency
[3]Wireless Application Protocol
[4]Application Program Interface
[5]World Wide Web

Acknowledgements

This project has taken a considerable amount of time and resources and I would like to acknowledge the help of all of those who have made the project possible. In particular I would like to thank my supervisor Peter Cornwell for his time, patience and guidance, and also for allowing the idea to be pursued originally. I would also like to thank Justin Cormack for his help, and my second marker Stuart Cox for his time and advice.

Further to these people I would like to thank the members of the Computer Support Group for their technical help in setting up various machines and telephone lines. Also, I would like to thank all of the many thousands of people who have worked on all of the Open Source projects without whose efforts this project would not have been possible. A list of these projects is given in Appendix A.

Contents

List of Tables

List of Figures

1 Introduction

This document is a report for the individual project “Implementation of a Generalised API for Application Control by Telephony Devices”. The project is an attempt to design and create an API (Application Program Interface) for applications so that they can be used by numerous different telephony services without specific knowledge of the capabilities of these devices by the application. Examples of these services, or access devices, include DTMF (Dual Tone Multi Frequency) tones from telephones, a WAP (Wireless Application Protocol) interface to mobile telephones and SMS (Small Messaging Service). The main difficulty with the creation of such an API is the fact that these access devices offer considerably variable levels of functionality to the user.

The need for such an API is justifiable. The use of fixed and mobile telephones is a common part of everyday life for the majority of people in the Western world, and is considerably more common than the use of personal computers. The use of these devices for input and output from computer applications, although now common, is relatively basic, with little standardisation. It is likely that the usage of these devices will increase in the future, particularly with the introduction of commodity hardware capable of handling concurrent telephone calls. By creating a standard generic API applications can be converted once, and with this change work with a large number of disparate access devices.

For the developer of a new application it gives a potentially greatly increased user base, whilst not creating too high an additional coding burden. A new application can also benefit from the advantage of non-recurring engineering allowing greater sophistication in the use of such an API. This project also explores the possibility that a user may wish to control an application from multiple locations regardless of the device that they have access to at that point in time - the fact that the user should not be limited to the case where they are physically in front of a computer hosting the software.

1.1 Aims & Objectives of the Project

The aim of this project is to create an API for application control by telephony devices, and to produce a number of example applications and separate device implementations. In other words the goal of this project is to abstract the input and output interfaces of an application to the extent that new access devices can be added to a system without any changes being necessary for the application. Thus applications do not need to be rewritten to cope with each technological advance, and devices only need to be added into this system once to gain from immediate access to all available applications. A significant part of this project is to look at how the API can be implemented so as to be flexible enough for use with many varied devices whilst being powerful enough to allow access to the functionality of the application. Some examples of these access devices are as follows :

- DTMF tones - Tones created by telephone key press, with audible response by text-to-speech synthesis
- SMS messages - Short textual messages available from mobile telephones
- Voice control - Spoken commands given by the user, with audible response
- WAP connection - A browser connection to a mobile phone or PDA

A number of example applications will also need to be created in order to evaluate the implementation. An example application would be to move a cursor around a screen, or to control a character in a game, from these devices. Application output should be provided to the user appropriately dependent on the access device used. With this API in operation, a user should be able to access all of the features of an application by telephone, rather than by having to sit at a terminal and use a keyboard and mouse.

The nature of the abstraction of the access methods should also allow for the implementation of access devices which are not telephony but network based, such as a web browser and an email message. Implementing these methods in addition to the telephony access methods will demonstrate the potential of this project and allow users more choice in the access of applications. It should be noted that the goal of this project is very much to prove this concept, rather than to produce an optimised solution which can undertake very high load from concurrently running applications. The API should however be able to support multiple applications, each with potentially multiple concurrent users, accessed from arbitrary access devices. It should also be able to sustain an acceptable interactive response for these users regardless of their access method.

Some possible uses for this API are at public information points, where a simple input device is preferable; as a tool for system administration to operate and control a system remotely; and as a method for members of an audience to participate in a large project, or vote on an issue, where the cost of providing input devices for multiple users may be prohibitive.

The aims of the project were originally set out in the project proposal[1] in November 1999.

1.2 Overview of the Project Progress

A formal view to the progress of the project was taken and resulted in a systematic development of the required components. In order to meet the stated aims, the following steps were undertaken :

1. Initial Design - The project was split into four separate parts, which were designed independently:

   (a) Design of the API - Consideration of issues such as the services required, the semantics of the needed operations and the language and systems to be used.
   (b) Design of a central service - In order to communicate with the access devices and the applications a central service is necessary. The main issues to be considered are how messages should be passed and how much state information should be held by this service.
   (c) Design of a number of access device methods - This included design of an interface for input and output between DTMF tones, a WAP browser, SMS messages, a telnet connection, email messages and a web browser.
   (d) Design of a number of sample applications using this API - This included an application to move a cursor around a grid, a presentation package and an application to navigate a 3D world.

2. Specification - The interfaces between the four areas were specified precisely, and the requirements for each of the four sections were elicited.

3. Implementation - These four areas were separately implemented, mostly in C, but using Perl where appropriate. Wherever possible existing systems were used to provide the basis of the functionality.

4. Testing - The implementation from each of the four parts were then tested as a single system

5. Evaluation - The system was evaluated through comparisons to existing systems were possible and through qualitative analysis of the commercial viability of the implementation.

On completion of the project this report was produced to document the decisions made in the above steps and to comply with the required regulations for a project report in the Department of Computing at Imperial College[2].

1.3 Overview of this Report

This report fully describes the project undertaken. However in order to control the length of the report, the reader is on occasion referred to a bibliographical reference if particular details are required in an area.

The report is split into five main sections :

1. Introduction - This gives an introduction to the project, its aims, an overview of the work undertaken in the project and an overview of this report.

2. Background Research - Analysis of current projects available in this area and an attempt to define what is the "state of the art". It also contains

analysis of the design issues for implementation of a possible solution to the problems raised, and a discussion of the technologies behind each of the proposed access device methods. Further, possible extensions to the basic design requirements are proposed.

3. Implementation - Discussion of the implementation choices taken and the software that was developed. Detailed reports are given for the implementation of the API and the implementation of the daemon which manages connections from users along with a sample application which uses the API and analysis of each of the implemented device access methods. In addition an example is given for converting an existing application to use the API and the subsequent development of a dynamic-link library which allows more effective use of the API by third parties is discussed.

4. Evaluation - Evaluation of the success of the implementation in meeting the needs discovered in the background research is given in quantitative and qualitative terms. The system is compared to other available products.

5. Conclusions - Analysis of the successes and failures of the project, and discussion of the advances made. A discussion of the commercial viability of the implementation is also given. Possible extensions and further work that could be undertaken are then discussed.

In addition to these main sections there is a glossary and bibliography of references at the end of the report, along with a number of appendices. These appendices contain :

- Appendix A - A list of the software and hardware used in the project, including URL in order that the reader may investigate the packages and tools further.

- Appendix B - A sample use of the produced library. A simple application which moves a cursor around a text grid is given in full to show an actual use of the telephony API library that was implemented. This example demonstrates how a developer could add this API to their own projects.

- Appendix C - A reference point URL for the reader to obtain the final version of the library produced, and also to download all of the deliverables associated with this project, including this report.

This document has been created in the LyX typesetting system which operates in the LaTeX environment.

2 Background Research

The project came about in response to the need to allow access to information and control systems from remote or mobile locations using everyday equipment. However, enabling this on a case by case basis for an application is difficult, time consuming and does not enable future devices to be used without changing the application. This background examines the steps taken to design the API by looking at the various components of the system, and gives references for further reading on the technologies used. This section tries to consider the current "state of the art" in this area by looking at currently available products and tools. From this the shortcomings of these products can be analysed, giving a base for the production of a new generalised API.

The rationale behind the project is that if a simple API can be developed as an interface between applications and these devices, then the task of adding each separate device is an operation which only needs to be undertaken once rather than on a per application basis. The simpler and easier that the API is to use from an application standpoint, the more likely it is that application developers will be prepared to support the API. Likewise another design goal is that the ability to add new devices is as simple as possible.

Overall then the project splits into :

1. The theoretical design of the API architecture
2. Implementation of this design
3. Implementation of at least one device "driver" which interfaces with the API
4. Implementation of at least one example application which interfaces with the API

However before looking at the actual design process, it is important to look at the motivation for the project in a little more depth, and to see which other projects or research in this area have been undertaken.

2.1 Motivation & Currently Available Products

Analysis of available research papers and extensive searching on the Internet has failed to uncover any other projects attempting this kind of generalised API for telephony devices. However, the API can be looked at from two other points of view - from the device side (considering a particular device such as DTMF tones) or from the application side (considering a particular application which has been designed for control by telephony devices). From these view points a number of systems can be considered as providing at least some control functionality.

Although many gateways exist between various telephony devices and other systems, for example between SMS and email, these do not give application control, nor operate in real time. The most similar case to be considered is in the use of a DTMF telephone to control and navigate a user through a menu system. However, in this case the application has been specifically created for this interface, and is dependent upon it. There are a number of cases where remote access has been used via the world wide web or by connecting to the machine using an application such as telnet, however these are also specific to a particular method and do not scale to a general case. As these methods are simple to implement on top of a basic API though, and are very useful for testing and debugging purposes as well as for demonstrating the functionality of the project, they could be employed here as exemplar services.

From the point of view of the application, it is common for the developer to create the application without the knowledge of the exact mouse or trackball that the user owns, just the idea that a pointer will be used. However, the situation in this project is a considerably broader approach where the aim is to minimise the knowledge necessary from the application developer about the access method, as this allows for a greater breadth of potential devices. A high proportion of applications are currently created tied to a keyboard / pointer interface due to the fact that this is available, rather than because it is necessary. It is almost automatic for any developer looking to create applications for the X Windows system to use the built in event model, which is based around KeyPress and ButtonPress events. In many applications, a simple cursor based movement suffices (eg. some games, reading a document, navigating a menu system etc.) whilst in others the user is simply looking at yes / no responses or checking status (eg. checking that a system is still alive, answering a questionnaire etc.).

A major aspect of the design phase is to make a decision on the basis of whether the application itself can benefit from the knowledge of the type of access device that is being used at that point in time, or whether the API implementation should make this completely transparent. The advantage of providing the application with this information is that it may be able to give output tailored to the access device. However, with this increase in functionality is brought an increase in complexity for the design of the API and complexity in the use of the API by the application. It is possible that the use of this feature if present can be optional for the application. In this case the complexity would only be there for applications that wish to make use of features specific to a particular access method.

Thus the motivation for proceeding with this project is that a large number of users who do not have access to a computer at a moment in time or in a certain location could still be using their applications with the devices available to them. Furthermore, if the API is designed successfully then adding this functionality should be very simple for application developers.

2.2 Design of the API

The API itself must be compatible with the major sets of access devices set out in the project proposal. Thus the capabilities to control applications are limited by the particular capabilities of these devices. In order to design the API itself, each of the possible device cases needs to be examined in order to determine how they could provide input and output to an application. After these have been considered, a base set of functionality can be established which can be used to consider various designs for the API itself.

For reference whilst considering these access devices, and the design of the API itself, an overview of the proposed architecture for the system is given in Figure 1.

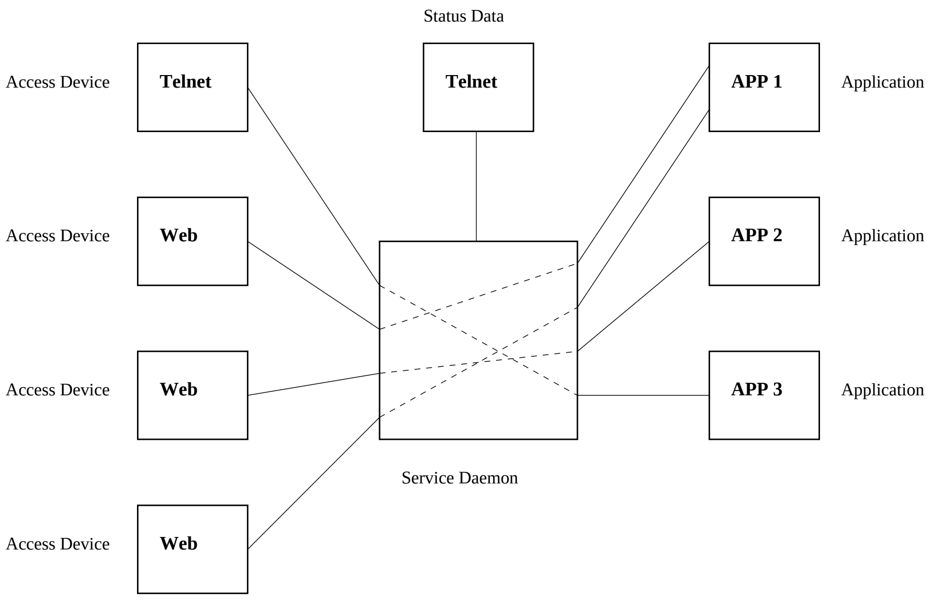


Figure 1: Diagram of the Proposed API Architecture Design

This Figure shows how a service daemon maps users who are connecting to the API from various access device methods to a set of applications. Also, status information can be obtained from the daemon by a telnet connection. The important point to realise is that the mapping should only be dependent upon the needs of the users, and not due to limitations in the design of the system. Also, note that the first application has multiple users - the requirement for this facility is discussed in section 2.2.3.

2.2.1 Devices to be used with the API

In the following device sections, host refers to the machine running the application and the API. For each device there is an analysis of the capabilities it contains and a view of the current systems available using this device. Note that the set of devices considered for inclusion was chosen on the grounds that the device is:

- Readily available to many potential users - Although devices with a small user base may well be used with the API, the priority was to focus on an implementation that can be used by as many users as possible, whilst keeping the necessary amount of work manageable.

- Capable of input and output - Devices incapable of input such as a pager, or only capable of output such as a speaker, are inappropriate targets for an API designed for users to control and receive feedback from an application.

- Capable of a usable level of interactivity - The API is designed for real time usage and thus a system such as conventional postage is inappropriate due to the delay which is unacceptable to the user.

- Capable of autonomous usage - The API is designed for embedded use without operator control. Without this the API would have it's scalability limited to the number of operators present, and this would considerably increase the costs of an actual implementation of the final product.

- Capable of initiating and terminating the connection - The user must be able to start the session by sending some form of request, the API should not have to poll users to see if they wish to send any requests to the system. This is due to the fact that a polling approach would scale very poorly - it is likely that for many applications the user may only wish to access the application very infrequently. e.g. A system administrator checking the status of a system. Likewise considerable load could be placed on the system if the user could not terminate sessions, and these had to timeout or were left open. Another consideration from this point is cost - network bandwidth and telephone calls are not free, and costs to the user should be minimised if the API is to become popular.

2.2.1.1 DTMF (Dual Tone Multi Frequency) Telephone This device is a standard POTS (Plain Old Telephone Service) telephone supporting tone generation by use of the numeric key pad. The device has 12 distinct inputs given by 12 unique tones (characters 0-9, # and *), no display and output is via real time audio. A continuous connection is established for communication, with negligible communication delay. Full specifications of the DTMF decoding interface can be found in the "blue book"[3]. An example of this interface is a telephone banking service which requires the user to enter details such as bank sort code. Mobile telephones can also be used via this method, but note that the frequency range available is more limited due to the GSM specification[4]. In order to communicate with such a device a telephony card can be used in the host computer servicing the connection in order to interpret the tones generated by the user and to respond to the user with audible output through the telephone.

This output should take the form of a voice response, which may be generated via text-to-speech synthesis on the host computer.

There are a number of systems in current use which take advantage of the DTMF tones. An example is the telephone banking system run by Halifax Bank[9] into which the user enters their account number, sort code and PIN code with the system responding with balance and transaction information. Note that in this case the responses are often pre-recorded rather than generated on demand. It is also of note that the specific software that exists for some telephony cards to provide these services is not generalised to other access methods or even to other manufacturers telephony cards.

2.2.1.2 Voice Recognition This uses the same international phone system as with DTMF, but instead of inputting information via the numeric key pad, voice commands are given. This requires voice recognition techniques on the machine servicing the call, as well as the ability to interpret the incoming audio in some form of analog to digital conversion. The facility for this conversion is included with many telephony cards, but the actual interpretation of this data is not, and is left to the designer of the system. Note that it may be required for the system to cope with issues such as background noise, accented speech, muffled sound and ambiguous responses.

An example of a system in current use is that of the Odeon cinema group[10] which has a system for booking tickets by telephone. In this system the user states by voice the cinema they wish to visit, and the system is able to choose the most likely cinema from a known list. Thus the user can pro-actively choose the destination rather than having to choose from a set of supplied options. This is used as it gives a shorter response time to the user than having to select a cinema with a DTMF choice from a very long list. The advantage this holds is that it provides a solution to one of the main user complaints about automated telephony services - that the user is stuck in a pre-defined set of choices which limit the interactivity available, and force the user to wait to access the function they desire. This is even though the majority of systems do allow the user to access a function directly if they know the commands. A feature of note from the Odeon system is that the system acts for confirmation of all information supplied by voice, in order to reduce errors. Although this increases the time of call for the user, the time is still far less than if the user did not have the option to choose cinema by voice command.

2.2.1.3 SMS (Short Messaging Service) Message The vast majority of modern mobile telephone systems allow the transfer of textual messages using the SMS system (also defined in the GSM specification). These messages can be up to 160 characters in length, and can be sent and received by mobile telephones. There is no concept of a connection, each message is independent. No response time is guaranteed, however personal experience would give a common response of about 1-5 seconds. From the host side either a system supporting SMS messages on a telephony interface or some form of SMS gateway needs be used. The approach taken by most network operators in the use of an SMS Centre (SMSC) which transmits the message out onto the network, and receives messages sent out by the networks subscribers. The various network operators then peer with each other at the SMSC level. However, individuals are not accepted as a peering point, and in order to create some form of SMS gateway to the host computer some form of encapsulation through an actual mobile telephone is needed.

Research has not discovered any system in use in this country for controlling applications via SMS messages. However, there are many systems such as the one used by Orange[11] for delivering news about Formula 1 developments to their users. The SMS system is also used by the network operators of mobile phones for the delivery of microcode updates to mobile telephones - which is a use of the system for sending commands, but not at a user level.

2.2.1.4 WAP (Wireless Application Protocol) WAP offers a connectionless or connection oriented protocol stack. The page based approach is similar to the web paradigm. The process used is that a request for a page is given from the wireless device (eg. WAP capable mobile telephone) to a WAP gateway, usually tunnelled through a PPP connection to a modem. This machine then asks the web server given in the URL in the request for the appropriate WML page, and binary encodes the page on arrival. This encoded data is then sent back to the device, which decodes the information to display upon a screen of some kind. A full specification can be obtained from the WAP Forum[5]. A maximum page size of 1500 bytes is defined, and the assumption is made that pages are designed for use on devices with small displays such as mobile telephones. Thus the host machine needs to offer a web server capable of offering the WML pages which in this case will need to be generated on demand, through the use of the Common Gateway Interface or similar. WMLScript can also be used to offer additional functionality. The user may then respond by choosing a hyperlink or by entering textual data into a form. An excellent white paper on this emerging technology is available from AU-Systems[6].

Mobile telephones with WAP browsers are becoming increasingly widely available. Most WAP based services currently in use are cut-down versions of web sites containing up to the minute news, sport and travel information for the user. The WAP paradigm is being seen as exploitable with the use of highly personalised and localised data, with location being given by the triangulation of the position of the mobile telephone between base station points. However, this has a number of associated privacy issues. Although many WAP systems in use such as the one provided by Orange multimedia[12] allow the user to enter data (e.g. to pick up travel information on a particular road) there are no gateways currently available to map this information onto an application in a general way. In other words each use of an application such as travel information is currently written on a case by case basis.

2.2.1.5 World Wide Web A web based interface allows output to be presented as a page in a web browser. The page is formatted in HTML, possibly with the extension of Java / Javascript. The use of HTML forms and hyperlinks to other pages allows input to be received from the user. Apart from the differences between WML and HTML and the fact that no assumption on display size is made with HTML, this method is analogous with the previous WAP service. The host needs to run a web server as specified in RFC[7] 2068, which is capable of generating HTML pages in real time through the use of CGI techniques.

As an established medium of document delivery and application development there are a vast array of systems with web interfaces in use. The majority interface with applications by means of a CGI program running scripting languages such as Perl, Java Servlets or Active Server Pages. An example would be a bookshop such as Amazon[13] which allows the user to order books over the web - the web front end connects through a CGI interface to a transactional database. The CGI interface is well defined and used - although if a user wishes to control an application which has already been created or designed for the X Windows system or for a text terminal a considerable amount of programming is needed at the CGI level on a per application basis.

2.2.1.6 Email Electronic mail could also be used as an interface to the API. This is analogous to accessing the application via an SMS message, except that there is no limit of 160 characters per message built into the system. As with SMS there is no guaranteed delivery time, but in most cases the timing is of the order of a few seconds. The host needs to run a mail server as specified in RFCs 821 and 822, and an email address is necessary for receiving messages. The arrival of a message to the system will need to cause some program or script to operate and communicate with the API. This script will then be able to send a message back to the recipient.

A number of systems exist to take commands via email, with response given in a return email. An example of this is Ftpmail[14] which wraps an FTP session inside a set of email messages. However, there is no current map of this technique onto a generic input and output mechanism for applications, the mapping has to be made specifically, and a service to receive email messages has to be ran on a per application basis. Thus if a number of these service were to be ran concurrently, each would require separate filters and analysis of the messages. An example of the crossover between email and SMS technologies is that a number of gateways exist such as Genie[15]. These can be used to send an email using an SMS message, or vice versa. The fact that these sites have proven popular is a demonstration of the fact that users wish to access computer and telephony based services regardless of the device which they have at that time. One of the common uses of Genie is for users to send themselves email reminders through the SMS function of their telephone. This saves the user from having to carry a diary around with them, and eases the transition of the data into other applications.

2.2.1.7 TCP Socket By using a TCP socket connection to the host, textual information can flow in both directions at any time, as this gives a full bidirectional connection. There is no limit on length in this access method. In order to transmit a message a new line command is needed. The host will need to offer a TCP socket for incoming connections to attach to on an unreserved port, as per RFC 793.

The telnet application (or the more secure derivative ssh) is widely used for remote access to a machine. By running the same terminal as with a local login a user can seamlessly use the machine remotely. It is also possible to tunnel X Windows connections over this link. However, in order to gain mouse control remotely the user must run some kind of X Windows server on the local machine - the actual application itself cannot necessarily be controlled from the telnet client. And of course the user requires access to a computer with a telnet client. Thus in many ways the aim of this project is to extend the ability of having access to an application from a remote telnet session to the ability to access the application via each of the above devices, and to add access via a tenet client to applications which rely upon pointer input. This would enable control where a TCP socket connection is feasible, but the display cannot be exported.

2.2.2 Analysis of the Device Capabilities

From the various capabilities of the devices, a set of restrictions and requirements for use can be derived. A table of the capabilities is given in Table 1 in order to compare the available functionality on a per-device basis.

Access Device	Input Values	Input Length	Output Values	Output Length	Connection Continuous	Reply Delay
DTMF	0-9,#,*	1	various	unlimited	Yes	<0.1s
Voice	various	various	various	unlimited	Yes	<0.1s
SMS	ASCII	160	ASCII	160	No	~5s
WAP	ASCII	various	ASCII	1500	No	~2s
Web	ASCII	various	ASCII	unlimited	No	~2s
Email	ASCII	unlimited	ASCII	unlimited	No	~5s
TCP	ASCII	unlimited	ASCII	unlimited	Yes	<1s

Table 1: Capabilities of Devices for use with the API

From this table a set of minimum capabilities can be defined for the API to be capable of supporting all of these access device methods :

- 12 distinct input characters with one character limit for input (DTMF) - In order to be usable on a standard telephone the set of unique input keys is limited to the set (0,1,2,3,4,5,6,7,8,9,*,#).

- Input as distinct as quality of voice recognition (voice) - If voice input is used then in order for voice not to be a limiting factor at least this set of 12 inputs must be distinguishable uniquely.
- 160 character output limit (SMS) - Each message has an output limit of 160 characters in order to be able to use an SMS message interface. Note that some characters may be needed to hold state information depending on implementation, and thus the actual specification for number of characters for output in the API may be lower than 160.
- Possible communication delays (SMS, email) - The fact that there is no upper bound on delivery time for these methods may limit the real time ability of the API. I.e. The communication is asynchronous.
- Network delay / failure (web, email, telnet) - In general these methods are packet switched over public systems, which also gives no upper bound on delivery time. However if a route from the host to he user can be established over a private network with known quality of service then an upper bound may be established. The network should route packets around a failure but if the failure is on a critical (no other possible route) section of the path then the network which is beyond the control of the application may fail.
- Connection-less devices may fail to return to a message (SMS, WAP (in some modes),web, email) - In the cases where a permanent connection is not made it is more difficult to detect failure as there is no dropped connection. Note that failure detection in asynchronous systems is impossible, and thus the asynchronous nature of the system may need to be altered with timeouts or a similar mechanism.

The above requirements in many ways provide a specification for the interface - if the API is to be supported by all of these devices and if the application is not to be informed of a set of capabilities possessed by the access method. In order to satisfy these requirements, a simple textual format is needed for output to the devices (synthesised to audio for DTMF and voice applications), and this is limited in length to at most 160 characters. Acceptable characters are those which can be represented in all cases, which is the ASCII character set. Note however that some characters in the ASCII set cannot be vocalised for text-to-speech synthesis (e.g. a tab character) as they were included for the purpose of text formatting. For input from the devices, the two telephone cases considerably reduce the options to 12 distinct single character commands.

In order for the API to support all of these devices the base set of capabilities restricts the possibilities for the design of the API. Thus the majority of access devices will not be used to their full potential if this base set is used for the design. A decision has to be taken as to whether the API should allow the application to take advantage of the functionality of a subset of the supported devices. This could allow multi-character input for example, or output of greater than 160 characters. However, the aim of the project is to focus on the readily available telephony cases, rather than for example optimising the API for telnet, over which many applications can already function. The possibility does exist for expanding a base design at a later date to add the ability to inform the application of the capabilities of the access method. However, for the API to show the full power of the abstraction of input and output from the application the initial design is based on the underlying approach that the application has no knowledge of the feature set of the access device. Also, in order to allow negotiation of capabilities within the API, the design of the API would become more complicated, violating a central aim for the design. Likewise, it would make use of the API by applications more complicated if they were to take advantage of this feature.

2.2.3 Analysis of Likely Application Needs

In most ways, the interface to the application is determined by the interface to the devices, as the semantics for the devices are already fixed, whereas the interface to the application is totally flexible at the design stage. Simplicity has been identified as the key, allowing as high a number of possible devices as possible, and also enabling an easy implementation at an application level. This is taking the reasonable assumption that developers are more likely to include an option for the API if this takes little time yet enables a large number of users to access the application via a large group of disparate devices.

Thus a textual (string) based communication between application and API can be accepted for input and output. At this stage in the design process it is tempting to say that a distinct command (input from the device) could be interpreted as a simple integer. However, this is dangerous as it is an implementation detail, and is an unnecessary reduction is possible functionality. For example, although a DTMF phone can only give 12 commands, it is possible that the API could interpret these as a special case and that from an application point it would be presented that all devices have a multiple character text capability. It is also possible that at a later date an expansion of the base API design would have the ability for an application to be informed on the capabilities of the access method. This may include the ability for multi-character input support. In order for the API to be developed over time, as few restrictions as possible should be imposed at the base design level.

There is another key point which affects the semantics of the API - the ability to support applications which have multiple users simultaneously. Many applications can support multiple users simultaneously, and the API needs to allow multiple users to access an application during a certain period, using either the same or different access methods. An example of this would be a game played on a large display screen which could have a number of users present in the audience playing simultaneously either as a team or against each other. The API also needs to be able to support multiple applications simultaneously if it is to scale to a usable level (a host should not be limited to a single application at a point in time). The limitation on users should come from the hardware requirements of the host and the hardware needed under each access method, rather than the API itself.

The questions which are specific to the application side of the API are the semantics of how a connection should start, how much state information is held (by the application and the implementation of the API), how it should close a connection, and how it should deal with failures and errors.

2.2.3.1 Application Initialisation Applications to which this API may prove useful fall into two groups, those which will run on demand when a connection comes in and close on completion of the users commands, and those which run constantly and need to be able to connect and disconnect users continually.

For applications which are to run on demand with a connection to the API the API needs to be able to initiate the application upon a request from a user. Thus the API will need to have access to some form of table stating which address under which access method should initiate which application. On startup of the application, it then needs to "connect" the data streams between the device and the application, providing translation as needed. Thus the implementation of the API will need to accept a connection from a device, start the application, and then communicate with this application. Once the application is setup, messages will need to be transferred from the user to the application through these connections. Possible translations include voice to text for a voice interface, and stripping of control characters from a TCP socket connection.

For applications which are already running, the API has to contact the application with the information that a new user wishes to connect, and then join the data streams as before. Thus on a request from a user to use an application, the implementation of the API will need to be able to check the existence of the running application, and communicate with this application if it does exist. Thus the application must be ready to accept new connections at any time. As before, the role of the API implementation is to be ready to accept connections from all devices at any time, and then to pass these requests on in a standard format to the application.

2.2.3.2 Application State Information Whilst a connection exists between the API and the application, the API will need to maintain a table of not only which device is connected to which application, but also which user this applies to from an application point. Note again that it is possible to have multiple users, applications and access methods, with an arbitrary mapping between these. The API may well also need to keep track of some form of timeout for each access and application for error handling. From an application point the only state that needs to be held is that a number of users are connected to the application using the API.

The actual state information that needs to be held is thus dependent upon the nature of the connection. If the connection is a permanent specific connection (e.g. a telephone link over a phone line) then the connection is determined by the unique resource being taken. However if the device does not hold connection state, then it may be necessary to embed state in the messages themselves. For example with email messages it may be possible to hold identifying data in the headers of the message in order to uniquely identify the message. If this were not done then it would be impossible to differentiate which message was part of which application connection if a user with a single email address was using two applications concurrently. Thus the address of the sender cannot be used in isolation to identify communication in a connectionless environment. A combination of state held in the implementation of the API and state held in the actual messages is necessary.

2.2.3.3 Closing an Application Connection As it is possible for the case to occur where an application which was started by the API will need to continue processing after the user has disconnected, closure of the application cannot be the responsibility of the API. However, in all cases it is the responsibility of the API to cleanly remove any state information held on disconnection with the user and to be able to handle the case where an application terminates unexpectedly, whilst users are still connected.

In the case where the application allows user termination the API must allow for a user command to be interpreted by the application as a close connection message, with the application then responding to the user with a final message to inform the user the operation was successful, and responding to the implementation of the API that it has completed processing, and that no more communication with the user will take place. Note that the user may terminate a connection (e.g. by completing a telephone call) before the application has informed the API of completion. A possible solution to this is to combine the message which informs the user of completion with the message which informs the API.

For the case where the application will continue processing after the user has completed their usage session, the application can take the same steps to inform the API and user of completion. However in this case the application does not then terminate, but continues to be open to new connections through the API. The application may or may not have multiple users at any one time.

2.2.3.4 Error Handling & Failure Detection

Due to the nature of holding state on many open connections for a number of devices and applications of various types, correct handling of failures and errors is imperative if the API implementation is to be useful in practical cases. A number of possible errors may occur, and will need to be dealt with by the API implementation :

- Application errors - An application may fail unexpectedly. In this case the API implementation will need to inform the users and close their connections.
- User errors - A user may input data when this is not expected by an application, or input data that the application is not expecting. The API will need to hold at least the latest input from the user until the application is ready. However, it is the duty of the application itself to cope with input if it meets the required input set for the API.
- Device failure (broken connection) - If a device fails by breaking a connection this can be detected by the API and dealt with by informing the application of the failure, and then letting the application deal with this, possibly in a similar way to dealing with a user connection termination request.
- Device failure (connectionless device) - A lack of communication from a connectionless device such as a telephone sending SMS messages can mean either the user has stopped using the system, or that the system is running correctly and that the user is taking a large amount of time to come to a decision. To deal with this the API may need to define timeouts on connections, possibly on an application by application basis, as applications may have widely different expected response times.

2.2.4 Internal API Design

The API is very much defined by the needs of the devices and applications as described here. Given that the requirements analysed so far are met, the actual internal design of the API is very much open to the implementer. The API implementation should be able to :

- Takes setup data regarding type of connections, access method / address and the application these relate to
- Accept bidirectional connections from each type of device
- Initialise applications as needed from user requests
- Pass textual information to / from applications and devices
- Hold state information on all currently active connections
- Handle errors or loss of communication from any party

- The API should be scalable to the capabilities of the number of devices and applications available, if the host can support this load

There are also a number of general software engineering principles which should be adhered to in the implementation of the API, in order to maximise it's potential:

- The implementation should be portable between various host systems
- The code should be modular and well documented
- It should use minimal resources (e.g. only allocate necessary memory, listen on as few ports as necessary)
- The product should be stable under load and failures
- The implementation should be on interfaces which are defined to the extent that another programmer could create a separate implementation which would act in the same manner
- The implementation should maximise code re-use from currently available projects

In addition, there are a number of issues specific to the central API. It needs to run on a machine as a continuous daemon accepting connection requests from users via the device interfaces. Also, it would be reasonable for a suitably privileged user on the host machine to be able to obtain the state information from the API at any time, for debugging and logging purposes.

One purpose of the API that has not been considered in detail is that it should provide the end users of the applications with as simple a user interface as possible, even when taking into account the variety of devices in use. To this end, it is important that a user can ask for help at any point, and also that a user can end the interaction at any point. As we are limited to the use of the 12 characters on a standard telephone for commands, a reasonable suggestion is that the '*' key represents a command for help on available options at any time whilst the '#' key will always terminate a user's session. Providing semantics such as this reduces the learning curve for the user as they can expect standardised behaviour between applications.

More discussion of the chosen functions for inclusion in the API is given in the implementation section.

2.3 Extensions to the Basic Design

Assuming that the API is implemented successfully, there are a number of possible extensions that could be made in addition to the basic design. However, any changes to the API which add functionality are likely to do so at the expense of greater complexity, and possibly at the expense of reducing the available set of devices for application control.

Firstly, it would be possible to report back to the application the access method of the user, along with as much information about the user as the API can obtain. This could enable the creation of applications with “Avatars” which keep track of users and how they have connected. Similar in idea to cookies used in web environments, applications could then use this information to tailor their response on a user by user basis. The idea can be taken one stage further if a mapping can be made between a user and various access methods. This could enable the user to received their tailored application regardless of access method without the need for the user to provide identification information themselves. If a reasonable format could be found for defining an Avatar and the connection methods, it may even be possible to pass this between various different hosts running the API, as a method for automating the addition of a user.

Another extension could be to extend the API to support initialisation of a connection by an application. This could enable an application to contact a user if needed. An example of this would be a system monitoring application which tries to send an SMS message to a webmaster if a web server stops accepting connections. By storing data on various access methods per user a series of contact attempts could be made, in order of preference for the user.

It is also possible to extend the scope of the API to application and device connections. Although it is has been assumed that the host running the API daemon is also running the application and has access to the method for communicating with the device, this does not necessarily have to be the case. If a method such as TCP sockets is used for communication between the API and the applications, then it may be possible to make use of this service over a network. The advantage of this extension would be that the application can run on one machine, whereas the daemon can run on a separate machine with different hardware / operating system etc. Thus it could be envisaged that a few central servers could act as gateways between machines dedicated to certain hardware interfaces, and other machines hosting applications. The main issues with this enhancement are dependency upon the network, the powers of remote application execution and the security implications of open networks.

3 Implementation

The implementation of the API consists of a set of separate applications, which are then used together as a single system. This section of the report discusses the implementation of the core API design, the UNIX based daemon which implements the API, a sample application which uses the API and the implementation of drivers for the various devices discussed in the background. The subsequent conversion of the API implementation to a separate library is then detailed. There is then discussion of how to convert an existing application to use the API, with an example of a presentation package. This section is completed with a discussion of the expansion of the API to support applications with multiple concurrent users.

Please note that the focus in this report is on the features specific to this project. For example there is not an in depth coverage of the details needed to create a TCP socket for bidirectional communication. This is due to the fact that many works have covered these areas before and it would vastly expand the size of this report. Where possible references have been given if the reader would like to examine the area in more detail.

3.1 Core API Calls

The first part of the implementation for this project is to design and implement the calls or functions in the API which are used by applications, and which then communicate to the service daemon which holds the state of a users connection. This interface is actually part of the application, called by the application when it wishes to use the API. The use of this API gives the abstraction of the application's input and output.

Note that this section of the report is written in terms of the methodology of single user applications. The additional details necessary to enable multi-user applications are discussed in section 3.7. This is to enable the reader to gain an understanding of the entire system before discussing some of the more complex issues in detail.

The design of the API calls must meet the requirements set out in the background section. In addition there are a number of features which are desirable for the design of the API calls to hold :

- Rigidly yet simply defined - Ambiguity or complexity increases the difficulty for programmers to use the API with their applications. Thus the number of calls in the API should be as few as possible, in order to reduce complexity.

- Accessible - As the majority of UNIX based applications are written in C or C++ the API must be usable in these languages. However, the API design should not limit the user to a particular language, it should be implementable across various platforms.

- Consistent - All calls should follow a format which allows easy integration with an application by following a consistent design methodology. For example all calls should start with the same prefix to allow for the user to easily identify calls to the API in their code and to minimise namespace pollution.
- Correctness and repeatability - The implementation should match the initial specification. From this it holds that if an application repeats an operation, and if all other conditions hold equal, the result should be the same.
- Status information - Each call should return information as to the success or failure of the call to the application, in a standard defined manner.

Applications require calls to initiate a connection, receive input from the user, return output to the user, and to terminate a connection.

When a connection is initiated by a user, the service daemon needs to either contact the application if it has already started, or to initiate the application if not. In order for the application to communicate with the daemon, an initial call to the API must be made. This call is to lodge information with the daemon about the application, and to give the first piece of output to the user in order for them to know that they have connected successfully. To meet the requirement for output to be a textual string of at most 160 characters, a text string is necessary to pass this argument.

In order to communicate with the API, the send() and recv() system calls were used over a TCP socket link. This has the advantages over a UDP socket of providing reliability in the communication, and thus reducing the work necessary in the API and in the daemon. It also allows more easily for the possibility of having the application on a different machine to the service daemon, as the TCP socket interface is reliable across an IP networked environment. A highly readable introduction to TCP sockets and the send() and recv() primitives is available from the California State University[16].

The first call available in the API is thus :

```
int tele_api_init(char* init_msg)
```

The "tele_api" part is a standard prefix to differentiate calls to this API. The call takes a string for an initial message to the user. In order to avoid a C buffer overflow for security reasons the implementation of the API reduces the length of the init_msg to a maximum of 160 characters if it is greater than this, and provides a null terminator. The integer returned is used to inform the application of the status of the call made. On success the call returns a positive value. If a problem has occurred a negative value is returned. This follows the return types used by the standard C library calls.

Of note is the fact that the tele_api_init() function call does not take any information to identify the application to the daemon. This issue is discussed in detail under the implementation of the service daemon, however a brief explanation is that the unique process identifier of the application is embedded in the actual message sent to the application by the tele_api_init() function. In the case of each of the calls the actual message sent by the implementation of the API is not just the message provided by the application.

Once a connection has been established, the requirement is for input from the user and output to the user. The noted input requirement was for a character in the set (0,1,2,3,4,5,6,7,8,9,*,#). Thus it would be possible to pass input to the application in the form of a character or an integer (the ASCII or unicode representation of the character). However, if a string is used then it may be possible in a later extension to the API to allow for more information to be passed to the application then just the command character. An example of this is where devices capable of a response containing multiple characters may give a multiple character input value to an application. This would require the application to have previously established service levels for a user with the daemon.

Therefore the second call available is :

```
int tele_api_input(char* command)
```

This follows the standards given so far for a namespace of tele_api and to return an integer success value. An output command is therefore needed for use in response. The syntax for this is similar to the tele_api_init() function where the application provides a string and this is output to the user. As with tele_api_init() the string is reduced to 160 characters if it is greater than this amount. This call is :

```
int tele_api_output(char* msg)
```

Control of the application is now complete, with the user able to provide input commands and the output from the application is returned to the user in the appropriate form for the user's access method. Eventually the user may wish to terminate the connection. As discussed it is the duty of the application to provide a termination method, and then inform the API of the termination along with a final message for the user to confirm the success of the operation. Thus the final call necessary is :

```
int tele_api_close(char* end_msg)
```

As with the first three calls the standard format of namespace choice, string value and return value are adhered to.

With these four calls, an application can completely abstract all input and output to the API. The application is of course free to provide additional input and output methods if this is desired. The use of the API does not preclude the use of other event models and other input and output schemes. All that is necessary is for an application to be ready to receive input from a device connected through the API if there is one at any particular time.

In order to make these calls all an application needs to do is to include the C code for the calls. This block of code contains a #include compiler directive to a header file. This includes the necessary definitions for the standard C library's that are necessary for the calls to communicate over TCP/IP with the service daemon, along with forwards of the tele_api functions which allow the user to make the API calls before the presence of the functions in the code. This also allows the user to place the calls in a separate file if desired.

It is worth noting that a further call was added to ease the implementation. Due to the use of character pointer based strings in the code a convenient method is needed to clear the memory allocated by malloc() calls. A call is added which sets a portion of memory to zero (null) starting at the given character pointer. The amount set is a number of bytes given by the argument length :

```
int tele_api_clear_mem(char* data, int length)
```

Although there is no requirement for an application to make use of this call it is highly recommended that the application developer makes sure that all strings used with the API are cleared before use. This function may be used for this purpose. As with the other calls a negative value is returned on failure, with a positive value returned on success.

3.2 Service Daemon

To service communication from the API calls, a central service daemon was devised. This daemon runs continuously and holds all the information regarding the connection of an application to it's users (herein referred to as state information). As discussed above the daemon communicates via TCP sockets with users and with applications. Note that for the majority of user access devices (all apart from users connecting through a TCP socket using telnet) the connection to the user is not direct, but through another application or script. These applications provide the daemon with the commands from the user, and also take output from the daemon and provide this output to the user. This enables the design of the service daemon to focus on state information and on communication techniques, leaving the process of managing the disparate technologies separate. Another feature of the daemon is the ability to service requests for status as to the current connections open with the API. This function is vital both for the purposes of debugging for the application and access device developer and for logging system usage by system administrators. Note that the code for this daemon, and each of the following access device implementations, can be downloaded from the URL given in Appendix C.

The actual interface to the daemon is solely through the use of TCP sockets. This enables all communication to pass over networks if desired and gives a guaranteed level of service through the delivery of packets in the order they were sent. It also maps cleanly onto a telnet connection which uses TCP sockets. In order to act as a server with TCP sockets the daemon "listens" on a number of public interfaces, or ports. Ports are given by an unsigned 16-bit value (0-65535) in the TCP header. Port 0 is reserved, and ports from 1-1024 are only available for recognised services (eg. telnet on port 23, ftp on port 21 etc.) run as a privileged user on a system. Ports from 1025-65535 are available for applications to use as needed. The implementation of the daemon listens on port 5000 for telnet connections. It also listens on port 5003 for TCP connections looking for status information. Ports 5002 and 5003 are used for communication to applications and access devices. These values are arbitrary and defined in the header file - they can be changed as needed, but of course the same changes must also be made to the connecting programs. When a connection is received on a port, it is either dealt with immediately or the connection is moved to another port which is held open. This frees the listening port for new connections. Thus the visible ports are only busy for the minimal amount of time.

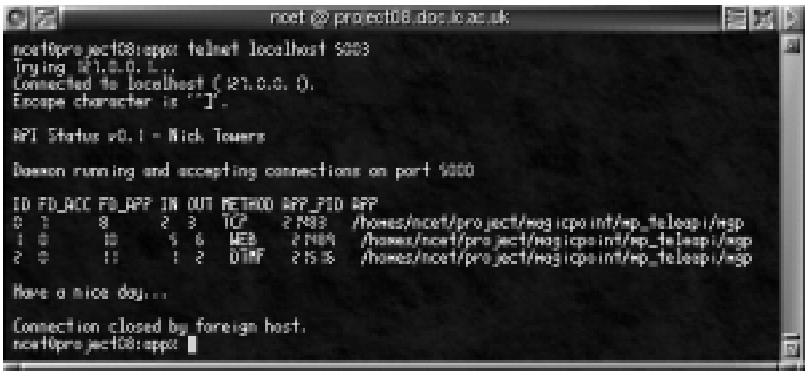
The implementation of the daemon raised three main issues which are worthy of discussion. Firstly, how much state information should be held by the daemon, and how should the daemon cope with errors and dropped connections. Secondly, which is the best way for the daemon to manage listening on a number of ports simultaneously, given that the basic recv() primitive is a blocking synchronous call. Thirdly, given that each TCP send() and recv() call is inherently synchronous and may take a significant amount of time given the access methods in use, should methods such as threading or forking the daemon execution be used to provide scalable levels of response from the daemon.

3.2.1 Managing Connection State Information

The daemon holds the information data on the state of each application currently using the API, and each user of those applications. If a connection is initiated by a user, a memory structure is initialised to hold this data. The application is referenced by the unique process identifier of the application, and the file descriptor of the open communication socket. For access devices, the file descriptor is used when a connection is held open (eg. for telnet) and where this is not possible an identifier for the connection is passed in the messages themselves. Thus with an email message for example the connection identifier is held in the message header. The daemon is able to distinguish the access method used on incoming connections by a single character field present in messages received from the access methods (except in the case of telnet connections which are identifiable as they use a separate port).

The daemon also holds information on the number of messages passed between an application and it's users for status data to be available. An example of the information available for status is shown by the capture in Figure 2.

The screen capture shows the response gained by using telnet to access the



```
ncet@project08:app% telnet localhost 5003
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.

API Status v0.1 - Nick Towers

Daemon running and accepting connections on port 5000

ID FD_ACC FD_APP IN OUT METHOD APP_PID APP
0  7      8      2  3   TCP    21483   /homes/ncet/project/magicpoint/mp_teleapi/mgp
1  0      10     5  6   WEB    21484   /homes/ncet/project/magicpoint/mp_teleapi/mgp
2  0      11     1  2   DTMF   21516   /homes/ncet/project/magicpoint/mp_teleapi/mgp

Have a nice day...

Connection closed by foreign host.
ncet@project08:app% 
```

Figure 2: Screen Capture of Daemon Status Information

status port 5003. The daemon responds with the information that it is running and accepting TCP socket connections from users on port 5000. It is also accepting other connections but these are not listed as they cannot be changed once the system is in operation. The output then shows that there are three current connections. The first, identified by 0 (ID field) is a TCP socket connection (METHOD field) to the magicpoint application (APP field - see section 3.6.1 for more information on this application). The application has the unique process identifier 21483 (APP_PID field). This connection has passed 2 messages as input (IN field) and has sent 3 messages as output to the user (OUT field). The file descriptors of the application (FD_APP) and the access method (FD_ACC) are also given for debugging purposes. The second connection is to another instance of the magicpoint application to a user via the World Wide Web access method, and the third to a separate instance by a user connected from a telephone. Note that the file access file descriptors for these methods are both zero. This is due to the state information being held outside of the daemon due to the lack of permanent connection to these devices (see the sections on these devices for more details).

The previous screen capture shows the daemon running successfully. However, there are a number of issues concerning the handling of errors and closure of connections if an error or failure occurs. Firstly, the daemon correctly handles the case where an internal error occurs or the daemon is sent a signal to shutdown by the operating system. It achieves this by "catching" signals sent by the signal handler through the C signal library. A discussion of UNIX signal processing is given in the Introduction To UNIX signal Programming[21]. Secondly, application or user errors are caught through the use of timeout values for each connection. After a connection is unused for a specified amount of time, it is closed by the API. The implementation of this feature is discussed in section 3.2.3 as it is closely linked to the use of threads for each connection. Were these errors not handled the daemon would eventually consume all of an available resource as the resources used by failed connections would not be freed.

3.2.2 Select() Communication Model

The daemon listens on a number of ports simultaneously, waiting for messages to be sent. However, this facility is not trivial to implement in C. Each port on which the daemon is waiting for a connection, and each port on which communication has already been established, may receive data at any time. In C, these ports are modelled as file descriptors, capable of use with the standard C send() and recv() primitives. A solution is to continually poll all of these descriptors to see if they have any data for delivery. However, this is a waste of the processor resource on the machine hosting the daemon, as for the vast majority of cases the poll will return a negative response. It is possible to insert sleep() statements into the poll loop, but this delays the response time of the API when a message is actually waiting.

A solution to the problem of checking multiple file descriptors is available in the select() system call provided in the standard C library. This call accepts sets of file descriptors to watch. The call sleeps whilst there is no activity, and then returns when one of the file descriptors has characters available for reading. The call can take any file descriptors - it is not limited to to socket descriptors. The main advantage of using this call is that the daemon sleeps not using any processor time until it is needed. The daemon is able to listen on all it's external interfaces through this single call. On startup the TCP socket file descriptors for the access ports are added to the set of file descriptors to watch. Subsequently, new descriptors created upon the launch of applications or incoming telnet connections are added to these sets.

The select() call also has the feature that it may timeout after a given time value - this is not used in the daemon due to the presence of threads for each connection (see the next section), but would be useful if there was a requirement to log status data periodically. Under Linux there is also a poll() call available which provides for more general handling of file descriptor events. This was not used in the daemon as the additional functionality is not useful in this case, and using the poll() call would limit the portability of the daemon to other UNIX implementations such as Solaris or FreeBSD. More information on the select() system call is available from Beej's Guide to Networking Programming - Using Internet Sockets[16].

3.2.3 Use of the Linux pthread library

Threads are a way of splitting the execution of an application into multiple concurrent paths. The main advantage of using multiple threads as opposed to multiple processes is that threads share the same execution environment as their parent. As threads share the same address space and global variables a context switch between threads is far less of a penalty than between separate processes, and the cost in system resources of thread creation and destruction is also far less. This also enables the various threads to update central memory structures, such as counters. Under Linux threads are implemented in a POSIX threads compliant C library entitled pthreads. In systems with a large number of users, it is possible to have thousands of individual threads running concurrently. In contrast, it is normally only possible to have a few hundred concurrent processes. Detailed information on the use and implementation of pthreads under Linux is available from The LinuxThreads Library homepage[17].

There are two main issues which benefit from the introduction of parallel execution threads into the service daemon :

1. Blocking on synchronous IO requests
2. Error timeouts on each connection

If the execution of the daemon were a single monolithic process, then it would not scale well under load. This is because the send() and recv() calls made to the applications and access devices are blocking calls, which means that they cannot be interrupted once made. Thus, if the daemon is receiving a message from a user who is connected over a slow network link, the daemon is not able to service any other communication requests until the original request is complete. Although this is not too serious given the short messages transferred in the API, it is much more serious if a connection fails whilst transmitting a message (for example a network failure), in which case the send() or recv() call blocks whilst attempts are made to re-establish communication. By splitting the execution into a thread per connection, each thread can block indefinitely without affecting any other connections.

Introducing threads also gives a much cleaner interface for using timeouts to check for dead connections. To implement this with a single select() call, a separate timeout queue would need to be kept for the active connections and the call to select() would need to timeout if a response had not been received by the timeout value of the head of the queue. Maintaining this queue would add complexity to the core function loop of the daemon, increasing the amount of time that the API is not able to respond immediately to requests. Through the use of threads the timeout can be held locally to each connection thread, leaving the main execution to wait in the select() call at all times. Once a timeout value is reached, the thread can destruct safely, removing the connection state from the central memory structures, and can terminate the application via a close message (#) as if sent by the user. This avoids the possible problem of using up the system resources available with application processes blocked waiting for input which never arrives.

Therefore when a connection is initiated by the daemon a thread is created for that connection. Any communication over that connection is handled by that thread, leaving the central parent process to listen on the publicly available ports for new connections. Concurrency issues are minimised by limiting global variable access to read only accesses or updates of logging counters. For these counters the order with which updates are made does not matter. This design mirrors the approach taken by most UNIX daemons offering TCP/IP based services.

3.3 Using Applications with the API

With the daemon implemented, an application was needed to show the API in use. The first application developed was an application which allows the user to move a cursor around a grid. It is discussed in the subsequent section.

From the point of view of the application developer, there are a number of items to consider when implementing an application which uses the API for input and output. These include :

- The event loop - The API is based on the premise that the application, once initiated, will be blocked waiting to receive input from a user in the tele_api_input() call. Thus the application needs a central while(true) or similar construct event loop with this call. If the application needs to process other events or execute other functions whilst waiting for input it may be necessary to perform these functions in other execution threads.
- Error handling - All calls to the API may fail and will show this by returning a negative value. The application should handle these errors gracefully. For example if the tele_api_init() call returns a negative value the application may wish to terminate as communication could not be established with the daemon. This could be caused by the daemon not being present on the host.
- Adding the code for the API - The code for the implementation of the API should be inserted into the application, and the header file for the API made available in the include path. Alternatively, the API can be used as a dynamically linked library (see section 3.5).

In addition it is necessary for the daemon to be able to start, or communicate with the application. This is achieved by editing the configuration file for the daemon, to give the path to the application and any command line arguments needed.

3.3.1 Cursor Application

The Cursor application demonstrates the facilities available in the API. The example of moving a cursor around the screen is analogous to the input necessary in many applications - for example navigating a menu structure or moving a player in a game. The resulting output which gives the location of the cursor demonstrates the speed of the response of the API, and the fact that the responses are generated on demand. The full implementation of cursor documented with explanation of the calls made to the API is given in Appendix B. A screen capture of the application in operation is also given.

The application provides the user with the following usage session :

1. The user initiates the application from their access method.
2. The application starts with the cursor at position (0,0). The user receives the string “Welcome to cursor, initial position (0,0)”.
3. The user may at any time receive help by entering the “*” command. The application responds with the string “Use the cursor keys to move around the screen, or # to quit. The application will respond with your position."
4. If the user gives any of the commands (1,2,3,4,6,7,8,9) the cursor is moved by one grid position in this direction. The cursor wraps around the grid if at the edge. For example, if the user enters “6” when at position (0,0) the cursor is moved right to (1,0). The application would respond with “Position (1,0)”.
5. If the user enters the commands “5” or “0” the cursor remains in the same square, and the application responds with output telling the user of the current position.
6. When the user wishes to end their session they enter the “#” command which terminates the application and their connection through the API. The application gives a final output through the API of the string “Hope you had fun in cursor" before terminating.

Note that the cursor application has no knowledge of the access method of the user, and that this information is not in any way necessary for the application to operate.

3.4 Access Device Implementations

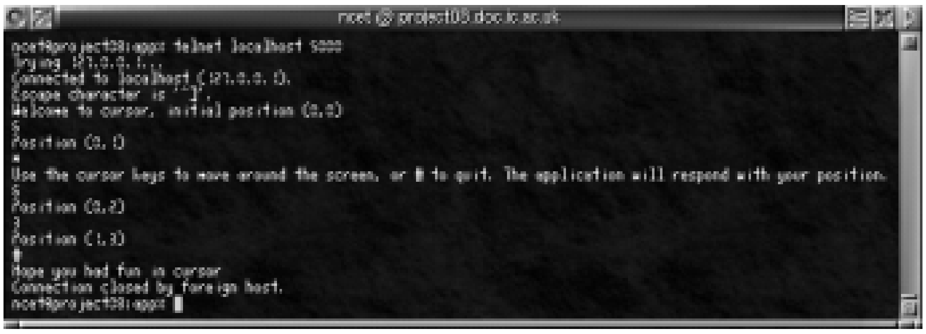
In order to make use of the daemon and the sample application, implemented access devices are needed for the user to connect from. The implementations for each proposed access method are discussed in the order that they were implemented - this order was determined by the potential difficulties in implementation. The source code for each of the implementations can be found by following the URL given in Appendix C.

The given methods are those implemented during this project. Other methods could be added by any developer at any time, without any changes being necessary for the applications in use with the API, or to the API itself. The one change that would be necessary id to update the daemon with the type of access method so that the method appears correctly in status information gained from the API. A possible extension could allow even this change to be unnecessary as access methods could identify themselves and their capabilities in communication with the daemon, rather than inserting this information into the configuration files and implementation of the daemon.

3.4.1 Telnet

The most simple access device implementation is for a telnet connection to the available TCP socket interface of the daemon. All of the implementation for this method is internal to the daemon. Once the user connects to this public access port, the daemon moves the connection to another port in order to free the port for other users to access. The connection from the user to the daemon is held open continually, and is referenced by the daemon using the unique file descriptor of the TCP socket.

The user inputs commands using the input device of the machine they are running their telnet client on. It is necessary for the user to append a new line after their input, as telnet buffers each line and only completes the send on a new line character. Output is displayed on the next line of the terminal, and this is terminated by a new line character giving the user a clear line to input the next command. Figure 3 gives a screen capture of the telnet access method in use accessing the cursor application.



```
ncet@project08:app> telnet localhost 5000
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Welcome to cursor, initial position (0,0)
s
Position (0,1)
h
Use the cursor keys to move around the screen, or # to quit. The application will respond with your position.
s
Position (0,2)
d
Position (1,2)
#
Hope you had fun in cursor
Connection closed by foreign host.
ncet@project08:app> 
```

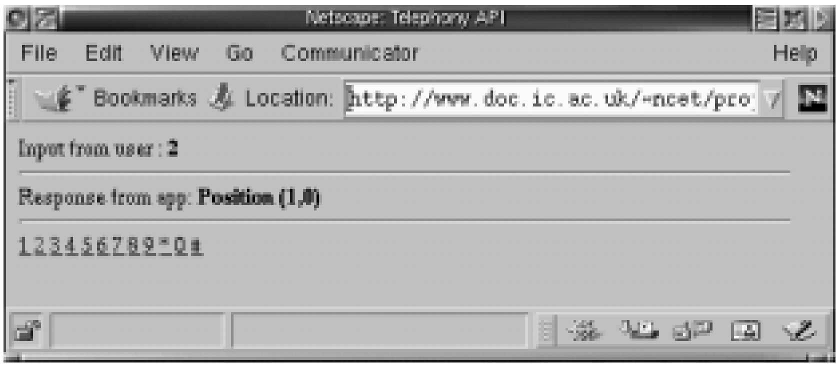
Figure 3: Screen Capture of Telnet (TCP Socket) Access Method

With the telnet access method the TCP socket connection is permanently held open. Thus errors or failures can be detected by a failure to be able to send() or recv() on the socket.

3.4.2 Web

The web interface was implemented through the use of a CGI script written in Perl[18]. This script is ran by the web server and the output of the script is the HTML code that is returned to the user. The web server used is the Department of Computing central web server[19] which runs Apache[20], with the mod_perl extension. The CGI and IO::Socket modules were used in the script to aid the implementation.

A user initiates a connection by accessing a URL on the web server which points at the Perl script. Input is given to the script as parameters appended to the URL, as with most CGI techniques. The script then communicates with the daemon over a TCP socket link. This link is only available for the duration of the individual request however, not the entire usage session, as the web server is connectionless in respect to the service daemon. A discussion of how error checking is undertaken for connectionless services is given in section 3.2.3. Thus once the script has received the return message from the daemon the link is closed and a HTML page generated from the applications response. This page contains a link for each possible user command, which point back to the Perl script. The actual state on the users connection is held in these links, as the single script is used by any user accessing any application. An example use of the web access method is shown in Figure 4.


Netscape: Telephony API
File Edit View Go Communicator Help
Bookmarks Location: http://www.doc.ic.ac.uk/~ncet/pro

Input from user : **2**

Response from app: **Position (1,0)**

1 2 3 4 5 6 7 8 9 * 0 #

Figure 4: Screen Capture of Web Access Method

One issue of note is that of cacheing. As the responses from the API are from a real-time connection and may vary on each call to the script, the resultant HTML pages should not be cached by the web browser. One possible solution is to insert a random string into the URL, so that the same URL is never accessed more than once, and thus web browsers do not show previous calls to the API. However, as the cached copies of the pages are never to be accessed again it is a wasteful usage of the browser's cache (and any other web cache the user is accessing) to cache the pages at all. A preferable solution used in this script is to add the HTTP header "Expires: -1d" which informs all caches that the content has already expired, so should not be cached.

3.4.3 WAP

The WAP solution uses the same Perl CGI script as the web solution. The only difference is that for a WAP browser the script returns a WML page as opposed to a HTML page. The script is able to differentiate users into WAP or web users by a parameter supplied in the access URL. In returning a WAP page the script returns the HTTP header "Content-Type: text/vnd.wap.wml" as opposed to "Content-Type: text/html" for a web page. In order for the web server to be able to serve WML pages the configuration of the web server has to have the text/vnd.wap.wml content-type setup as a valid MIME format. This process is described in the independent WAP FAQ[22] section 4.2.

The actual format of WML pages is considerably stricter than for HTML pages as WML is defined as an XML DTD[5]. However, the basic design and formatting features are very similar in the two languages, enabling an identical user interface to the web access method. The WAP specification does not specify how the pages should be shown on a particular device, but just which markup tags the device must support. Thus although all WAP compliant devices can use this access method, neither the application developer nor the maintainer of the perl script can state how the WML will appear on the user's device. For example the Nokia 7110 mobile telephone[23] places each hypertext link on a separate line. Although this makes usage of the interface more awkward as the user may have to scroll down a number of times to reach the desired input value, it does not in any way limit the functionality available.

The main implementation difficulty with the WAP access method was with the WAP gateway to be used as opposed to the actual generation of the WML. At the time of writing WAP is a very new technology, and few public WAP gateways are available, with none located in the United Kingdom domain. In order to facilitate debugging of the WAP implementation, and to save on external bandwidth requirements, a WAP gateway was installed on the host machine. The gateway used was a snapshot release of the Open Source WAP gateway project, Kannel[24]. By running the gateway on the same host as the service daemon, response time was noticeably improved, and the processing of each WML page could be logged.

A diagram of the WAP implementation is given in Figure 5.

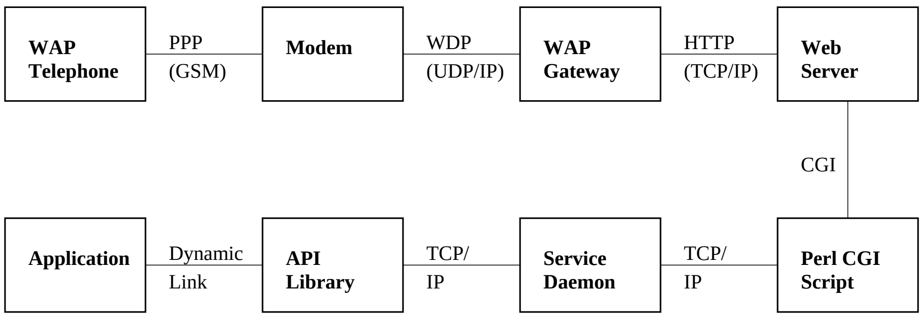
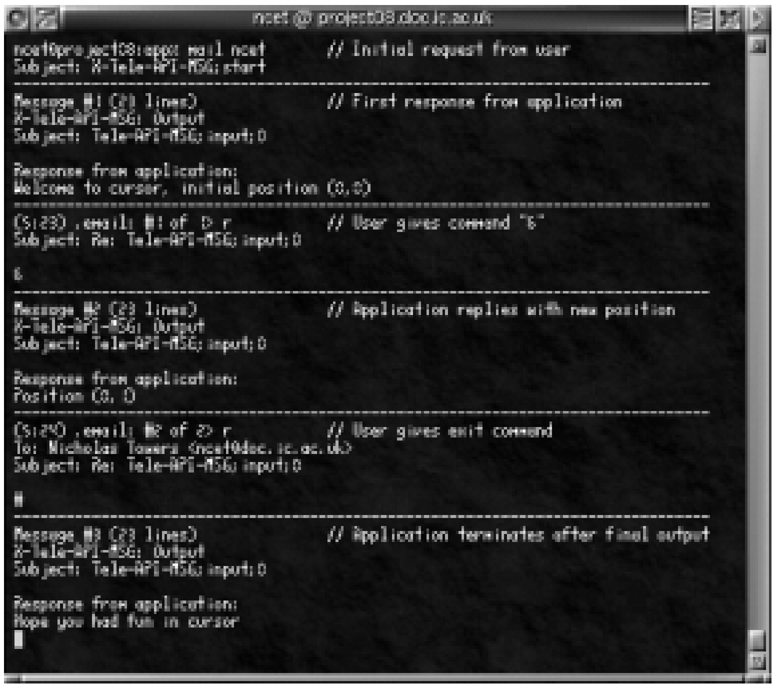


Figure 5: Diagram of the WAP implementation

Thus a user can connect to a modem in the Department of Computing modem pool with a PPP connection from their mobile telephone. All of the user's traffic connects to the WAP gateway over a UDP connection (using WDP). The gateway then requests the desired URL from the web server (with a HTTP request) which runs the Perl script to communicate with the service daemon. State information about the connection to the application is held in the URL itself. The Perl script generates the WML page which is then delivered from the web server to the WAP gateway. The gateway encodes the WML into a compressed binary form and this is then delivered to the access device over the modem link. The device then displays this page. Although the telephone has a continuous circuit based connection to the modem, the connection to the service daemon is connectionless through the web server.

3.4.4 Email

Although the response time for an email message can vary considerably, it is still a valid access method for many applications, and email is available to many users in situations where the other access methods are not available. The implementation of this access device is based on a similar Perl script to the web and WAP methods. The script is run on receipt of an email to the system giving user commands, and an email is returned containing the response from the application. An example usage session is given in Figure 6. Note that the screen capture has been edited to remove irrelevant message headers and output from the mail reader.



```
ncet@project08:apps mail ncet        // Initial request from user
Subject: X-Tele-API-MSG;start
--------------------------------------------------------------------------
Message #1 (23 lines)                // First response from application
X-Tele-API-MSG: Output
Subject: Tele-API-MSG;input;0

Response from application:
Welcome to cursor, initial position (0,0)
--------------------------------------------------------------------------
(S:23) .email: #1 of 1> r            // User gives command "6"
Subject: Re: Tele-API-MSG;input;0

6
--------------------------------------------------------------------------
Message #2 (23 lines)                // Application replies with new position
X-Tele-API-MSG: Output
Subject: Tele-API-MSG;input;0

Response from application:
Position (0, 1)
--------------------------------------------------------------------------
(S:24) .email: #2 of 2> r            // User gives exit command
To: Nicholas Towers <ncet@doc.ic.ac.uk>
Subject: Re: Tele-API-MSG;input;0

#
--------------------------------------------------------------------------
Message #3 (23 lines)                // Application terminates after final output
X-Tele-API-MSG: Output
Subject: Tele-API-MSG;input;0

Response from application:
Hope you had fun in cursor
█
```

Figure 6: Screen Capture of Email Access Method

The process of filtering incoming email is achieved using the Procmail system[25]. This enables any message sent to an email address with a subject line containing "Tele-API-MSG;" to be forwarded as input to the Perl script rather than be delivered to a particular mailbox. In order for a user to initiate a new connection the user sends an email to an address with the subject line "Tele-API-MSG;start". The perl script will respond to the user with a return email containing the initial output from the application. Sendmail[26] is used for sending email in this manner. From this mail onwards the state information is held inside the headers of the email messages sent, in a header entitled "X-Tele-API". It is preferable to encoding the information in a header as opposed to the subject or body of the messages as it is less likely to confuse the user with excess information, and is more difficult for the user to edit, mistakenly or maliciously. Once the user has received the first response from the application they can input commands by replying to the previous received email with the command in the first line of the body of the message.

It is worth noting that the Perl script executed is actually executed on the mail server delivering the email to whichever address is used, with the access permissions of the owner of this address. As all communication is over TCP sockets, this does not cause any issues for the implementation, but it does require Perl to be available for use on the mail server, and also that the mail server is capable of sending email as well as delivering email.

3.4.5 DTMF Tones

In order to implement a DTMF tone interface a telephony card was needed for processing telephone calls. The PIKA Inline-GTi card[27] was chosen for a number of reasons :

- Linux driver support - The PIKA card had support for the Linux operating system through the use of a C library and two kernel modules. Sample applications using this library were also provided to ease development.
- Multiple lines - The card supports up to 4 active telephone lines at any one time. This allows the implementation to demonstrate multiple users of the API simultaneously via a DTMF tone interface from one hardware card.
- Cost - The card was reasonably priced in comparison to the other cards available at around £500.
- International Support - The card supports European as well as US phone system standards.
- Available Software - The Bayonne[31] telephony server (previously entitled ACS) is available for the PIKA platform under Linux. This may enable some service level comparisons of the successfulness of the evaluation to be made. Note that the Bayonne system is however specific to DTMF phone systems, and does not currently support real-time TTS synthesis.

The PIKA expansion card uses an ISA system bus slot as found in a standard personal computer. The card is accessed by a provided C library entitled "MonteCarlo". This library accesses the card through the use of two Linux kernel modules which have direct hardware control of the card through /dev entries in the host filesystem. Although the DTMF application created now executes correctly using this library, is of note that a considerable amount of time in this project was lost obtaining functional versions of this library from the PIKA development team, and working around bugs and incomplete sections of their code.

A sample application (in C) provided with the library was used as the base for the implementation. This saved a considerable amount of time as the sample application provided included support for correctly initialising the card, handling calls and for playing audio files. The application was edited to change the central execution loop from being based on user input to an entirely reactionary approach determined by events occurring on the phone lines. This required changes to the event handler which used C call backs to update global variables in the application. By querying the value of these variables in the central event loop, the state of the phone lines can be established.

The DTMF implementation actually creates a number of separate processes. At initiation, the execution is split into a number of concurrent processes using the fork() command. This gives a separate process for each phone input line. The advantage of separating the execution into distinct processes is that as each process is a copy of the original it is not possible for the executions to interfere with each other, as is the case with threads. By separating out concurrency issues, the implementation can take a more simple procedural form.

A separate process is also created in order to run a server for text-to-speech (TTS) synthesis. The Festival system[28] created at Edinburgh University was chosen due to it's ability to run as a client-server application. Thus each of the processes handling the telephone lines can request TTS processing at any time from this server using the Festival client. This has considerable advantages over the option of executing a TTS application for every case of synthesis needed, as TTS systems require considerable resources and have a non trivial initialisation and loading period. Thus by running the Festival system the server can be started on initialisation of the DTMF application, and is then ready to service requests from the four phone line processes. This provides considerably lower latency in the TTS synthesis stage. The ch_wave application provided as part of the Festival package was also used in order to convert the output provided by the Festival server into a standard SUN audio data file with u-law encoding.

Thus if a user calls a number connected to the PIKA card, the application responds by accessing the service daemon over a TCP socket connection. The output response from the application is sent as text to the Festival system which returns an audio file. This is then played to the user. Commands from the user are given using key presses on the telephone, which are interpreted by the application as disparate DTMF tones and then sent to the service daemon as a command character. This process continues until the user gives a "#" command which terminates the application after it sends a final output message. Once this message has been played the phone line is replaced "on-hook" by the application, and is then available for the next user. All four lines may be used concurrently by separate users if required, and the users may be using different applications through the API. The mapping of a user to an application is by telephone access number. Note that the response time to the user of the DTMF method is largely dependent on the amount of TTS synthesis undertaken, and an extension to the implementation which cached previous responses or buffered the synthesis stage (so that some of the output is spoken to the user whilst text later in the output is still in the synthesis stage) may considerably improve the interactive response for the user.

As the DTMF application is run continuously, it can hold the state of a connection locally. This avoids having to keep a permanent TCP socket connection to the service daemon open, and preserves resources. It is not possible for the service daemon to know the status of the connection between the DTMF application and the user, but the DTMF application can inform the service daemon if a connection is dropped.

3.4.6 SMS Message

It has not been possible to implement an SMS access device interface to the API. This has been due to aspects beyond the control of this project, rather than technical difficulties. In order to send and receive SMS messages, a route to an SMSC is needed. However, the network operators in Britain limit access to their own customers, or to the other operators with which they have a message peering agreement. Although some operators offer services such as Genie providing email to SMS and Web to SMS gateways, these services do not offer a response time suitable for interactive applications - for example with Genie, no delivery is guaranteed, and delivery is only expected within 24 hours of sending the message.

The only option for placing a message into the SMS system is therefore to actually use a mobile telephone connected to one of the central networks. A mobile telephone is connected to the serial port of a machine, and is used to send and receive messages. Thus the architecture for an SMS interface would be to connect a mobile telephone to the machine, and then pass incoming messages to an application similar to the DTMF interface. By running continually, this application would be able to hold state on the connections mapped to users and their connections to the API. The telephone number of the incoming message can be used to make this mapping. This saves having to hold status information in the messages themselves, which is difficult with SMS as there are no user definable headers. Although it limits users to one application per phone at any one time, this is not a serious limitation as the user interface design of mobile telephones makes the control of multiple applications concurrently impractical.

At the start of this project in November 1999 the ability to use mobile telephones through the serial connection on a machine was designated as a part of the Kannel project to be completed in the near future. However, this work was not actually entered into the codebase until June 2000 and at the time of writing still only supports a limited set of mobile telephones, none of which are available to this project. No other projects were found that contained support for this feature. Thus although the design is complete, the timeframe for completion of this project does not allow for an implementation of an SMS interface. Had the progress of the Kannel system in this area been known at the start of the project, an implementation of this feature within the project may have been considered more deeply. The actual implementation now included within Kannel has shown however that this task is not trivial, and would have been beyond the scope of the original aims of the project.

3.4.7 Voice Control

The implementation of a voice control access method was carefully considered. However, an implementation was not completed for this project due to the difficulties involved. Although various voice recognition systems are available for Linux, they are mostly based on the principle of a user dictating a stream of text, from which the recognition system can use grammatical, syntactical and statistical approaches to determining the speech, as opposed to simple word detection. Conversion of these systems for the needs of this project were impractical in the given time frame.

The only package which offers some development possibilities is the CVoiceControl package[29]. This system is designed for the conversion of spoken input to UNIX commands, and does process individual words separately. Unfortunately, experiments with this system gave highly variable results dependent on issues such as background noise and the exact distance of the user to the microphone. One of the main issues is that the system was designed for use in a quiet environment with a high quality microphone with a high bandwidth, directly connected to the system. However, from telephones (in particular mobile telephones using GSM compression) the correct command was only received on a small fraction of occasions, with background noise creating many spurious results. As the time available did not allow for the considerable work that would be needed to solve these issues, it was decided that implementing a voice control method for this project would be impractical.

Were the CVoiceControl package (or another system) to have provided acceptable results then the system could have been implemented as an extension of the DTMF system, and possibly used interchangeably with it. Thus users would be able to either give voice or DTMF input at any stage. The speech recognition system would run on each phone line process, continually listening to the input given. If a DTMF tone is received, it would be passed along as before. If a voice command (audio above a minimum threshold) is received then heuristics could be used to determine the most likely command and then this could be passed to the service daemon. Of course, the issues of background noise and audio which cannot easily be recognised would need to be considered in a solution. Output would be returned to the user in the same manner as for the DTMF implementation.

3.5 API Dynamic Library

When the sample application, service daemon and multiple access device methods had been implemented, the focus of the project changed to how the implementation of the API could be most easily integrated into other applications, and how this process could be made simpler for application developers. Rather than simply inserting the code for the API calls into an application at development time, there are a number of advantages to being able to call functions in from an external library. With a library, the API code is compiled separately to the application code, and then associated with the application by the linking stage of compilation.

By extracting the calls to a library a number of advantages are present :

- Abstraction - By separating the code for the API from the application developer, both can be developed and updated independently. This also reduces the length of the code for the application.
- Security - As the code for the API is separate from the application code, the application developer cannot change the API implementation either mistakenly or maliciously.

In C there are two methods of accessing external libraries - either statically or dynamically. With statically linked binaries, the library code is inserted into the application by the linker at compilation time. However, with dynamically linked binaries the library code is located centrally on a host and linked into the application at run-time. The advantage statically linked binaries hold is that they are fully portable between hosts, and cannot be "broken" by the absence of another file on a host. Dynamic libraries have some considerable advantages over static libraries. As the object file is linked into the application at run-time, the size of binaries produced is considerably smaller, as they do not actually contain the code for the library calls. The compilation time is also subsequently reduced. Leading from this is also a memory footprint advantage - rather than loading a copy of the library code for each application linked against it, with a dynamic library it is possible to load just one copy of the library into memory and share this between all applications on the host machine. Thus the system can scale to a far greater quantity of simultaneously running dynamically linked applications than statically linked applications. Also, as the application is making use of the published interface to a dynamically linked library, the application or the library can be updated independently, without having to recompile both parts. For example if an update were made to the API implementation, as long as the entry calls were the same, all application programs using the API do not need to be changed or recompiled to take advantage of this update.

In order to use implement the API as a dynamically linked library the API must be compiled with the additional flags of "-c" to not link the code and "-fPIC" to produce code which is position independent. Were this not the case the compiler would specify exact locations of instructions, which would mean the code could not be dynamically linked into an application at arbitrary locations. The dynamic library can then be created by calling the compiler with the flag “-shared”. In order for use in the Linux library namespace, it is also necessary to use the naming convention of providing symbolic references to the library under the following scheme. The actual library is given the extension “.so.x.y” where the so refers to shared object, the x refers to a interface specification version and the y to the version of the library which implements this specification. Multiple releases of the same library can be installed on a system to support applications linked against each x release. For each x the latest available y implementation is used.

Once compiled an application is able to use the API by use of the “-L<location> -ltele_api_lib” flags to the compiler (where location is the location of the library). It is also necessary for the library to be present on the library search paths of the system or the developer. In order to allow the application to compile with calls to the library functions, the header file for the library must be included in the application. An example use of this method is given in the implementation details of the “cursor” application in Appendix B.

3.6 Conversion of Existing Applications

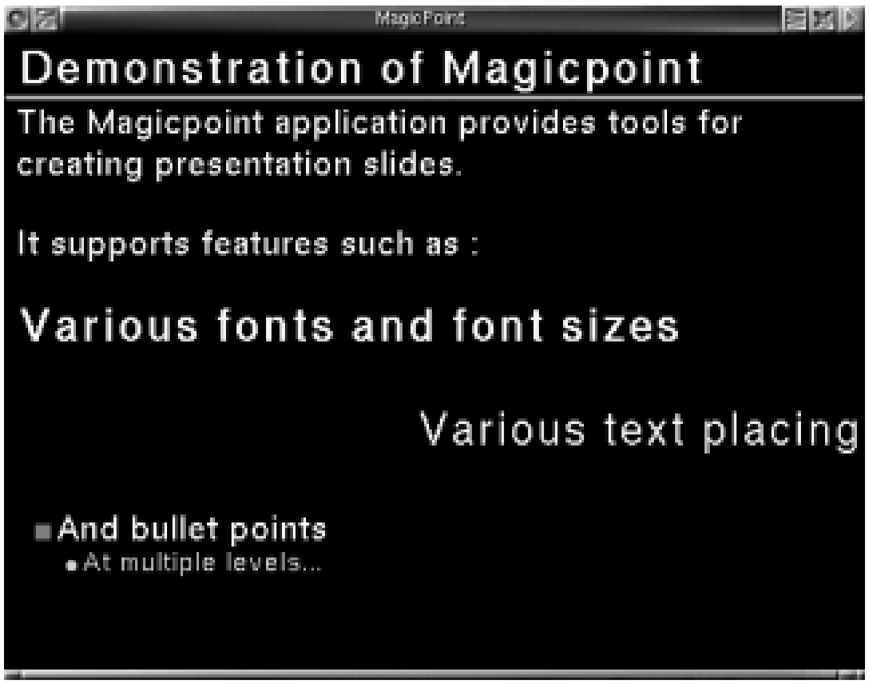
With the dynamically linked API library installed, the process on converting applications to use the API is simplified. The following steps are all that is needed for taking an application and converting it to use the API. Note that none of the original application functionality has to be removed as the operation of the application through the API can easily be switchable with a command line parameter or configuration file option. If this option is not present the application can function identically as before the API was introduced :

- Include the header file of the API and add the location of the API library to the Makefile.
- Exchange the central event loop (for example the XEvent loop in an X Windows application) with a do-forever style construct, as with the sample application discussed previously.
- Map the desired user input commands onto available commands provided by the API, and then on receipt of input call the existing commands.
- Map the output of these functions to a textual string, and give this as output to the API.
- Check for error responses from the API and either terminate or handle these errors.

It is obviously the case that the mapping of existing input and output procedures may be difficult in some cases. However, as discussed at length, this is necessary for all access devices to be able to control an application. A demonstration of the fact that the loss of functionality may not be that large is given in the next section, with an example walk-through of these steps.

3.6.1 Magicpoint Presentation Package

A freeware presentation package for Linux exists in Magicpoint[30]. This system takes in presentations created in a specifically formatted text file, and outputs presentations to the screen. The user may then give commands to change the page forwards or backwards through the presentation, or jump to a specific page. In use, Magicpoint is similar to the Microsoft Powerpoint application, but has key bindings based on the vi text editor, and does not have an integrated application for the creation of presentations. A screen capture of the application in use is given in Figure 7.



Demonstration of Magicpoint

The Magicpoint application provides tools for creating presentation slides.

It supports features such as :

Various fonts and font sizes

Various text placing

- And bullet points
  - At multiple levels...

Figure 7: Screen Capture of Magicpoint Application

There are a number of reasons as to why you may wish to access a presentation package through the API. The package currently only supports input from the mouse or keyboard of the machine displaying the presentation. This limits the position of the presenter to the surrounds of this machine in order for them to be able to change the page. With the use of the API developed in this project, the user can access their presentation from anywhere they wish - for example by a web connection from another machine in the room, or a mobile telephone (possibly with a WAP browser) from anywhere they desire. This enables the presenter freedom of movement around the room, or the facility to control presentations in any other location if desired. Although the need to run presentations in locations separate from the presenter are small, more are emerging. For example a presenter could give a presentation over a video conference link across the world, and use the API to control a presentation being displayed locally to a group at the remote site. This could be expanded to allow the user to control multiple presentations at multiple sites simultaneously. Another reason for using the API is that although the presenter is happy to stand next to a machine, it is impractical to send the video output of this machine to a projector at the rear of the hall. By placing the machine running the API next to the projector, the API allows the presenter to use a separate machine at the front of the hall connected by standard networking such as Ethernet.

The conversion of the application took about 15 minutes to complete, which gives an idea of the complexity involved. Firstly, a command line parameter (-Z) was added to give input control to the API rather than to XEvents. Thus if this parameter is provided, Magicpoint does not run the normal event loop "main_loop". Instead, the application calls tele_api_init() with a message for the user stating "Welcome to magicpoint - input 2 to start". This gives the user time to prepare to display their first slide, safe in the knowledge that the API has initiated correctly. From this point the user can give the "2" command to move forwards a page, or an "8" command to move backwards a page. If a "*" is input, the application responds with basic instructions, and a "#" ends the usage session. After each user input the application responds with confirmation of the page that is now displayed. This could be advantageous if the presenter is in a position where they cannot actually see the displayed output.

The mapping of the input from the API onto existing calls gave no problems as the commands to move one page could use the same code as was already present, without the XEvent references. For output, a simple page counter was held and this value was output as the variable in a string containing "Showing screen %d". The only functionality that could not be reproduced was the facility to jump to a particular numbered page (achieved in the original code by either a mouse click onto a menu or a keyboard pattern). However, this does not greatly affect the majority of users, as the vast majority of presentations are accessed sequentially. Were an implementation of this feature particularly desired it may be possible to implement a scheme were a command using the API means the next n following input values will give the page to jump to. For example the "3" command could be interpreted as the next three numbers sent as input will form a three character page reference which the application will jump to.

It is of note that the final code produced did reduce the level of encapsulation in the Magicpoint code, with similar implementations of the move page function in two sections of the code. However, if this is considered a problem it could have been solved by separating out the move page function into a procedure called by both execution paths. This would only take a few minutes to undertake, but would mean that it would be more difficult to apply the changes made to a later version of Magicpoint, as the version of the code with the API implementation would be further away from the original codebase. Ultimately the only ideal solution is to include the changes for use with the API in the released version of the package.

3.7 Support for Multi-User Applications

In order to allow for applications to have multiple concurrent users a number of additional details are required at the API design level. It is a requirement for the application to be able to identify which user has sent a particular message. Likewise, it is a requirement for the service daemon to be able to identify the intended recipient of a message. Furthermore, it is necessary to add and remove users whilst the application is executing in order to provide an acceptable level of functionality to the application. Otherwise, the users would have to wait for a group usage session to begin, and would not be able to leave until the session had completed.

A number of possible implementation approaches can be taken to solve these issues, however most require more complex function calls than those in the API. The fact that a full multi-user implementation is available without any more complexity in the API calls is a justifiable advantage over the addition of separate calls, as only one version of the API is needed to support single or multi-user applications, and thus there is only one version of the codebase to maintain. It also enables application developers to change from single to multi-user implementations with ease.

The implementation takes advantage of the integer value which returns status information on completion of each API call. This call result value was defined to give a positive value for a successful call, and a negative value for a failure. This design leaves an address space to signify that input is from a particular user - the set of positive integer values.

The maximum potential number of users for an application can be defined in the configuration file for the daemon, along with the access points for the application from each access device method. If no value is given, single user mode is assumed. For all applications, both single and multi-user, the user with identity zero is defined as the controller of the application (analogous to the superuser under UNIX systems). This is the user for which tele_api_init() is called, returning zero upon success. Single user applications continue as before calling tele_api_input() and tele_api_output(), until termination of the application when tele_api_close() is used.

For applications with multiple users however, once setup by the connection of user zero, new concurrent users (numbered upwards from one) are entered into the system with the tele_api_input() call. The application is able to tell whether it is a new or already existing connection by the integer returned. The application then responds, with the output directed by the API to this user. The user zero may access the application at any time as with the other users. If tele_api_close() is used in response to a user other than zero then the connection to this user is terminated, with the identity becoming available for a new user. If user zero calls tele_api_close() the entire application is terminated with the last output message returned to all users on their next connection through the API.

The advantage of this scheme is that it gives a central point for control of the application as user zero. More importantly than this though is that it gives application developers the ability to create multi-user applications without needing concurrent threads in their code - the application only has to perform one blocking tele_api_input() call. The application does not have to worry about queued requests or looking for connections from new users and established users concurrently, as these issues are covered by the API service daemon.

The implementation of this functionality in the service daemon is given through the use of threads for each connection as before. The difference with a multi-user application is that the structure held in memory describing the connection to the application also holds information on the current and maximum number of users, and a function is available to provide threads with an identity value not currently in use by the application. The daemon refuses connections to users if the application user limit has been reached with a single response message. These connection attempts do not reach the application.

An example application use of the multi-user implementation could be a game played on a large display screen. The administrator of the application would initiate the connection as user zero. From this point, players could join the game, play for a while and then leave indefinitely. When the administrator wishes to terminate the game, they can do this as user zero regardless of the number of users accessing the application a that time.

4 Evaluation

This project set out to prove the concept of designing and implementing a generalised API for application control by telephony devices. This section of the report attempts to evaluate whether this concept has been proven and to compare the results with other available systems. Some evaluation has already been given with the implementation of each part of the project and is not repeated in this section.

The background research section of this report discusses the need for a general solution to the issue of application control from ubiquitous access devices. By demonstration the API has achieved a solution to this problem - the cursor and Magicpoint applications developed can both be controlled by users accessing the application from a telnet connection, a web browser, a touch-tone telephone, a mobile telephone with a WAP browser or via an email message. This is possible without the applications having any knowledge of the access method used by the user. As the API has subsequently been implemented as a dynamically linked library it is also possible to extend the API implementation without any recompilation of applications linked to the API, as long as the external calls remain unchanged.

4.1 Quantitative Analysis of the Implementation

As a proof of concept implementation, it is difficult to quantitatively assess the applications produced, as there are no other systems available which provide the same functionality. Each part of the project implemented is therefore discussed qualitatively in the next section. However, it is possible to compare the results achieved with the original design goals. A table showing whether the anticipated features for the core API have been proven feasible by the implementation is given in Table 2.

Original Design Goal	Implementation Level
Abstraction of access device	Fully implemented
Uses C language	Fully implemented
Simplicity of design	Four API calls present
Flexible input to applications	12 distinct single character input values
Flexible output to applications	Up to 160 characters from the ASCII set
Multiple users per application	Fully implemented
Error handling	Calls return API status
Update API easily	Dynamically linked library implemented

Table 2: Evaluation of Features Implemented in the API Design

A similar analysis of the features implemented for the service daemon can be seen in Table 3. The configuration file change required in order to map any user to any application is also dependent on the access device in use. This is discussed qualitative analysis.

Original Design Goal	Implementation Level
Allow bidirectional communication	Fully implemented
Control applications on other hosts	Fully implemented
Facility to output state information	Fully implemented
Use select() IO Multiplexing	Fully implemented
Use threads for scalability	Fully implemented
Respond to UNIX Signals	Fully implemented
Map any user to any application	Requires configuration file change

Table 3: Evaluation of Features Implemented in the Service Daemon

Table 4 shows the features successfully implemented for each proposed access device method, and a value for the average response time (time for output to be returned after a command is given) found from each implemented device. This was calculated by averaging the response time from each device over at least 30 commands. These results are discussed in the next section under the headings for each device, including analysis of whether the delay is due to the API implementation or the message path to the device. Note that the variance in the result for DTMF is cause by a dependence on the length of output response received from the application.

Access Device	Device Implemented	External Applications & Code Required	Response Time
DTMF	Yes	C program, telephony card	<1s to ~5s
Voice	No	N/A	N/A
SMS	No	N/A	N/A
WAP	Yes	As Web + WAP Gateway	~5s
Web	Yes	Web server, Perl CGI Script	~1s
Email	Yes	Perl script, procmail filter	~4s
TCP	Yes	None	<0.1s

Table 4: Evaluation of Access Device Methods Implemented

Finally, Table 5 shows the features implemented in the sample applications developed for use with the API.

Application Feature	Cursor	Magicpoint
Dynamically linked to API	Yes	Yes
Usable from all access methods	Yes	Yes
Complete functionality with API	Yes	Most
Input values used with API	1,2,3,4,6,7,8,9,*,#	2,8,*,#
Output returned through API	Current location	Page update
Gives user help on request	Yes	Yes
Graphical library used	Ncurses	X11

Table 5: Evaluation of Sample Applications Implemented

Although these results are not absolute in most cases, they provide a base on which to determine the success of the implementations provided. Although a number of original goals were not completely fulfilled, this does not detract from the fact that the original concept of generalising the input and output to various disparate devices has been shown, without introducing unacceptable delays to interactive use.

4.2 Qualitative Analysis of the Implementation

Following is a qualitative analysis of each application and access device method developed along with the core API design, and the implementation of the service daemon :

- Core API Design - The API developed meets the all of the original specified design goals. It enables an application to abstract input and output characteristics to a generalised, device independent format. The goal of a simple and easy to use API has been achieved through the use of a minimal set of library calls, and though the use of a dynamically linked library for separation of the API implementation from the code of an application using the API. This enables updates to be made to the API without any necessary recompilation of the applications using the API. The design also enables applications the option of having multiple concurrent users through the API at any point in time. Further, the API design gives applications feedback on any errors that have occurred through the return values of the function calls. However, in order to support all the proposed access devices, the main limit of the API is that input commands are limited to single character commands from a set of twelve possible values.
- UNIX Service Daemon - The daemon implemented meets the requirements set with the exception of the requirement to map any user to any application dynamically. It provides a scalable threaded architecture for accessing applications through the API from a number of access devices. Full bidirectional communication is given for all access methods, and the state information held within the daemon can be obtained via a telnet connection for status collection or logging purposes. The daemon does not poll any of the various connections it holds, but multiplexes the connections using the select() C library call. The one requirement not met is that it is not possible for a user to access any arbitrary application at any time from all of the access devices - this mapping must be made for some access devices in the configuration files for the daemon and the access methods. The implementation of the daemon has proven stable under load, and has not failed after continuous periods of uptime.
- Cursor Application - The full functionality of the cursor application is available through the API, without any knowledge of the access device in use. A user can initiate the application remotely, and then move a cursor around a text grid. The application responds to a command by moving the cursor appropriately, and gives output to the user of their position

through the API. The standard use of "*" and "#" commands for help information and terminating the application respectively are upheld.

- Magicpoint Application - The Magicpoint presentation package has successfully been converted to use the API. The conversion process took about 15 minutes to complete, and enabled access through the API to the application. The main functionality was fully implemented, with just the ability to jump to a specific page not available with the API interface. A possible extension of the implementation to provide this feature is given in section 3.6.1. The user is able to change the page of their presentation through the API and receive an output response giving the page that is now being displayed as appropriate to their access method. For example with a DTMF tone interface the response is as speech.
- Telnet Access Method - The telnet access method is fully implemented allowing application control from any machine with a TCP/IP connection. Commands are given as a single character input, terminated by a new line. The application output received in response is displayed on the next line of the terminal. This response is almost instantaneous (providing the TCP/IP link is of at least modem quality). A mapping to the desired application can be provided by allocating a port on which the daemon listens to each available application .
- Web Access Method - The web access method is fully implemented as per the original design. A user initiates the connection with a stated URL, and inputs data through hyperlinks. Output is returned as a HTML page. Response time for this has operation has been measured at about one second, with the delay due to the processing of the CGI script on the web server (which is beyond the control of this project). A mapping of a user to a particular application can be made by giving a different URL for the user to access - which calls the CGI Perl script with different parameters.
- WAP Access Method - The WAP access method is implemented as an extension to the web method, giving full application control to the user of a WAP capable mobile telephone wherever they are located. The response time for the user via this access method was measured at about five seconds, with the application processing the command about two seconds into this period. On top of the delay introduced by the web server, the WAP gateway used considerably delayed the response. This delay is likely to decrease significantly as the Kannel WAP gateway project moves towards a full release version, to give a total response time in the order of about two seconds.
- Email Access Method - This is fully implemented with a Perl script receiving the email via a Procmail filter. A user sends email to a known address, and the application responds with a return email. The response time of

about four seconds is mainly due to the two email client systems tested, with about three seconds taken in the process of informing the user a new email has arrived, and being able to access that email. This is the case even though the user was expecting the message. Whilst this reduces the usability of the email access method for interactive response, the method is still useful as a method to fallback to if a user does not have access to any of the other methods at a particular location. As email is seen as an essential feature for personal digital assistants and internet access portals it has a very high user penetration, and consequently is available in situations where the other access methods are not. The mapping of user to application can be made by using a separate startup command in the initialisation email.

- DTMF Access Method - This method is fully implemented with the use of a PIKA Inline-GTi telephony card capable of supporting up to four simultaneous users. A user dials into a given access number and can give commands through the use of the telephone keypad. Output is spoken back to the user through text-to-speech synthesis. The response time for this method is heavily dependent on the length of the output returned through the API. If an empty string is returned, the response time is less than one second. However, strings of the maximum permitted length of 160 characters can take up to five seconds to synthesise - and also require more time to be played back to the user. Possible improvements to this response time are considered in the implementation discussion. The mapping of a user to an application can only be achieved by allocating a different telephone line to each application, and the user can choose application by the number they dial. However, this is not an optimal use of the resources available on the card, and it limits the number of applications that can be used with this particular card to four.

- SMS Access Method - This was not implemented due to the incomplete status of the SMS implementation in the Kannel project. However, such the design for such a system is given, and the expected response time would be in the order of five seconds, depending on network operator.

- Voice Access Method - This was not implemented due to the lack of an available voice recognition system with suitable features. After extensive research into this area it was decided that undertaking such an implementation would have detracted from the focus of this project. An implementation would have taken considerable time creating or adapting a voice recognition package for use with telephone systems. If a suitable system had been found the implementation would extend the DTMF system, with the voice recognition expected to add approximately two or three seconds to the response time.

4.3 Comparison to Other Available Systems

As previously stated, there is no other single system which implements a generic API for application control by telephony devices with which to contrast this project. This does not mean however that comparisons of parts of the system developed cannot be made to existing systems which fulfil a specific niche. With systems which provide only conceptually similar facilities, it is only possible to give a high level comparison of the usability of the systems. This can be done by making comparisons between such areas as response time, functionality, ease of use and convenience to the user. Later in this section a comparison is given between the DTMF access method implementation and Bayonne[31]. Due to the similarity between these systems a more in-depth analysis is possible.

The actual API itself cannot be directly compared to any other system as it is concerned with requirements not addressed by any other work. In contrast with the XEvents model used by the X Windows system a far higher level of abstraction is given, with the XEvent model the application accesses an input event from a specific device (for example a press of a particular button). The XEvents system is also considerably more complex, with a far steeper learning curve for an application developer. Many other APIs are tied closely to a particular operating environment (for example the Microsoft Win32 system) or to a particular access device (for example the Linux gpm console mouse server).

The service daemon implemented is specific to the API in this project, and therefore cannot be compared on that level. It is possible to comment on the behaviour of the daemon in comparison to other UNIX services, for example the standard web and ftp daemons installed under Linux. As with these services, the daemon implemented in this project supports the use of UNIX signals to inform the daemon of system status, for example a signal to exit when a system is shutdown. The implementation given does provide logging information via a TCP connection, but this is in a proprietary format as opposed to the standard method of using the syslog process. This functionality could be added to the daemon if needed. It is also possible to compare the resource usage of the API service daemon in comparison with existing systems. Due to the use of the select() system call, the daemon does not use any CPU time when inactive (except when timeout values are reached). This is mirrored in most ftp and web servers. When in use, the daemon never consumed more than one percent of CPU usage on the test machines, considerably less than for the web and ftp servers which are used for bulk data transfer.

The sample applications given provide very general services (movement of a cursor or displaying a presentation) which are representative of many applications. Their use of the API makes them unique however. Although implemented solely for demonstration and testing purposes, the converted Magicpoint package has many potential uses, as discussed in section 3.6.1. The ability to control presentations from anywhere with an arbitrary access device is a demonstration of the power of the API, which cannot be achieved with any other system available. Also, the fact that it took only fifteen minutes to convert the application for use with the API provides a comparative value from which the cost of converting other applications can be more accurately estimated.

The access devices implemented can each be compared to specific examples of a device controlling an application, for example those given in the background research. The telnet interface provides considerably less access to a system than a terminal connection, but does provide the ability to control graphical applications in situations where only a textual interface is available. The web interface provides a greater level of control than most available systems, with the lack of customisation of the user interface by the application the main tradeoff. In order to provide a generalised interface through the API a comparatively basic user interface is given which does not take advantage of the features available to the extent of most web sites. The WAP interface on the other hand offers a similar user interface to most other sites, due to the presentation limitations of the WAP model. The application control given is more powerful than that of any other currently available system - with the majority of WAP sites in use at the moment tied to a specific database query engine. The email interface likewise provides considerably more functionality than other systems. Although an email interface to a system is not as easy to use as a web or telnet connection, it is available from more locations. It is also of note that using email as a control system gives automatic logging of all commands and responses conveniently into the users mail folders.

The DTMF interface can be compared at quite a low level to the Bayonne system. The Bayonne system is an open source project developed for the same PIKA telephony cards as the DTMF interface for this project, and which uses the same MonteCarlo API. The Bayonne system is designed for systems such as voicemail and customer response lines, where a user dials into the system and navigates a series of menus, receiving spoken feedback. However, although the Bayonne system is extensible it was inappropriate for use in this project due to the lack of facilities for gaining full control over the telephony card, and for the fact that it uses pre-recorded voice responses as opposed to dynamically generating them from application output which is done here. Although using pre-recorded responses limits the Bayonne system in functionality, it provides a considerably shorter response time for the user. Bayonne and this DTMF implementation use the DTMF tones in the same manner for input. With the Bayonne system however, no standard is suggested for receiving help information or terminating a connection. In many ways this is an advantage for the DTMF implementation given here, as this gives a common interface for users which decreases the time it takes for them to learn to use a new application. Testing the systems together, the only noticeable advantage the Bayonne system holds is with the speech feedback. As well as a faster response (measured at up to 400% faster with a 160 character message), the samples are actual human recordings rather then a synthetic voice. This is more pleasant for the user. In order to counteract this, a possible extension of the API implementation could be to allow applications to provide voice samples for at least some of the output responses to the user.

5 Conclusions

This section summarises the overall conclusions drawn from this project and discusses the possible commercial viability of the implementation produced. It also describes how the project could be extended with further work. The key conclusions that can be made are :

- A generalised API for application control by telephony devices has been implemented and evaluated successfully, despite the highly disparate functionality of many of these devices.
- With the API developed in this project it has been shown that the access device, whether telephone or computer, can be abstracted from the application. This allows a single access device implementation to function with every application linked to the API and likewise allows a single application implementation to function with every implemented access device method.
- Most applications do not need to have knowledge of the exact capabilities of the input and output method in use by the user.
- Although access devices such as a telephone have limited input capabilities in comparison with a personal computer, these capabilities are sufficient for the control of many applications, especially where visual or audible feedback is available.
- There are a considerable number of applications to which users desire access from wherever they find themselves, and with whichever device they have access to at the time.
- The abstraction layer introduced by the API implementation does not result in unacceptably increased interactive response time for the user. With the exception of the text-to-speech synthesis required for the DTMF access method, all significant delays in the application response to the user are outside of the API implementation in the message delivery path to the access method.
- The implementation of the API as a dynamically linked library gives greater encapsulation of the API and significantly improves possible commercial usage.

It is also worth noting that the availability of open source tools for many tasks enables a considerably higher rate of application development - without the existence of many of the external applications and utilities used, the implementation discussed here would not have been possible.

5.1 Commercial Applications of the API

The implementation of the API as a separate dynamically linked library, although not an originally stated goal of the project, provides for possible commercial applications. As a separate library it could be provided in binary only form, with full documentation of the calls available. It is feasible to do this without providing any of the original source code. The implementation of the library, service daemon and access methods could be packaged into a standard format for installation on a host, such as the Redhat RPM format. Application developers could then include calls to the library in their code, and enable access to their application from a number of disparate access devices. This facility, as discussed in detail has considerable justification and can bring a number of benefits to an application. These benefits, and the work saved by using the implementation provided by this project, give real value to the work, and thus potential commercial application.

There are however some issues that would need to be overcome for a commercial launch of the API as a product based on it's current state. The implementation is reliant on the presence of a number of external packages, and these would need to be included in a commercial offering. For many of these applications a large amount of configuration work is required, and it also the case that the license agreements for each of these packages would have to be carefully analysed. Thus in order to produce a system which could be successfully sold commercially, a considerable amount of work would have to go into a central installation and configuration utility to setup and administer the API. It would also be necessary to increase the scope of the access device implementations - for example the current DTMF interface is limited to the PIKA Inline-GTi telephony card. If this limitation was not removed it would dramatically cut the potential market for the API.

A further change is that it may be necessary to port the implemented code to Windows NT to increase the potential market. As all of the code is written in ANSI C or Perl this is a feasible transition to make. For the API to be offered as a commercial product the security implications of the system would also have to be considered. It may be necessary to increase the obfuscation of API state information passed in messages for example, in order to reduce the possibility of malicious users trying to gain access to systems through the API. For the ability to trace all accesses through the system it may also be required to implement logging through the syslog daemon.

Overall, there certainly exists a potential market for an API such as the one developed during this project and the steps to produce a commercial offering are now well defined.

5.2 Possible Extensions

A considerable number of extensions could be made to the work undertaken in this project. The three main paths of progression that could be taken are :

1. Extend the API architecture with additional functionality, and re-implement
2. Improve and extend the existing API implementation
3. Develop this API implementation for commercial usage

There is of course overlap between these paths. Almost every aspect of the implementation of each of the access device methods, and the API itself could be extended, refined and improved considerably. This is to be expected considering that the aim of this project was to prove the concept of such a system rather than to create an optimal implementation at every stage. Thus the following suggestions for possible extensions to the project do not form a complete list, but are those that would lead to the greatest increase in the level of functionality offered by the API :

- Completion of SMS and Voice access methods implementation - The design for these methods is complete, and implementations could be achieved in a relatively short period of time if appropriate tools can be found. This would complete the aims given in the original project proposal.
- Track users with "Avatars" - User access information could be tracked in order to gain information on usage patterns. This could extend across access devices, leading to the creation of an information and application portal for a user derived from their previous usage. The actual user identity could also be passed on to the application, by the API. This is particularly relevant with mobile telephones where a user is likely to connect from various locations but the accessing telephone number remains constant.
- Initiate user contact - Users could be contacted pro-actively if needed - for example using the mobile phone number of a system administrator to alert them of a problem with their system. Multiple methods could be attempted, for example making a telephone call could be tried first, then an SMS message sent if the telephone call attempt failed, and an email sent if this also failed. This would also give a better demonstration of the value of making services available from multiple access devices, dependent upon availability to the user.
- Package the API implementation as a system - In order to offer the API as a commercial product, the various applications implemented would need to be packaged together as a single system with a flexible installation and configuration utility.

- Provide full syslog logging - Conforming to the UNIX logging daemon syslog would enable full traceability of accesses through the daemon to be logged in a standardised form.
- Implement more complex sample applications - The sample applications discussed here are both relatively simple in design. In order to demonstrate the power of the API more fully, a complex and highly featured sample application is needed.
- Extend the API design - As discussed at length it would be possible to provide some feedback to an application about the level of capabilities of the access device in use. This would enable the application to tailor it's response in some cases. This however brings a burden of increased complexity. Likewise support for threaded applications to access communications from multiple users concurrently may be added in exchange for greater complexity in the API.
- Embed an application chooser in the service daemon - It may be possible to increase the functionality of the service daemon to provide the mapping from access device to application. This would allow the DTMF implementation, for example, to map any input telephone line to any application.
- Allow applications to "dock" with the service daemon - If applications could add themselves to the API this would remove the need for application setup information in the configuration files of the service daemon. For example a web based interface could allow applications to be added and / or removed from the set offered by the daemon, without losing any currently held connections by having to restart the daemon.
- Implement the API library in other languages - Although the library implementation is in C, there is no reason why it could not also be implemented for Perl or other languages. This would increase the number of applications which could take advantage of the API.
- Analyse the security implications of the API - If the API were to be used to control machines as a privileged user, or to applications with sensitive data, it would be necessary to analyse the security implications carefully. The API implementation given here assumes a user will not maliciously try to insert spurious or altered messages - this assumption cannot be made in all circumstances.
- Improve the response time from the DTMF access method - The response time of the implementation developed is dependent on the length of the output string given by the application. Numerous schemes such as cacheing previous responses and buffering the text to speech synthesis could be used to improve the response time for longer output strings.

Glossary

API Application Program Interface

CGI Common Gateway Interface

DTMF Dual Tone Multi Frequency

DTD Document Type Definition

GSM Groupe Speciale Mobile

HTML Hypertext Mark-up Language

HTTP Hypertext Transfer Protocol

IP Internet Protocol

ISA Industry Standard Architecture

MIME Multipurpose Internet Mail Extensions

POSIX Portable Operating System Interface

PPP Point-to-Point Protocol

RFC Request For Comment

RPM Red Hat Package Manager

SMS Short Messaging Service

SMSC Short Messaging Service Centre

TCP Transmission Control Protocol

TTS Text-to-Speech

UDP User Datagram Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

WDP Wireless Datagram Protocol

WML Wireless Mark-up Language

WWW World Wide Web

WAP Wireless Application Protocol

XML Extensible Markup Language

References

[1] The original project proposal :
http://www.doc.ic.ac.uk/~ncet/project/proposal.html

[2] Imperial College Computing project guidelines :
http://www-ala.doc.ic.ac.uk/~ajf/teaching/Guidelines.html

[3] DTMF specification : CCITT Blue Book, *Recommendation Q.24: Multi-Frequency Push-Button Signal Reception*, Geneva 1989

[4] GSM specification (for purchase) : http://www.etsi.org/

[5] WAP forum (includes specification) : http://www.wapforum.org/

[6] WAP white paper : http://www.wapguide.com/wapguide/Auwap.pdf

[7] RFC Documents : ftp://ftp.isi.edu/in-notes/

[8] RPM Package Format : http://www.rpm.org/

[9] Halifax Bank : http://www.halifax.co.uk/

[10] Odeon Cinema Group : http://www.odeon.co.uk/

[11] Orange plc Group : http://www.orange.co.uk/

[12] Orange Multimedia WAP services :
http://www.orange.co.uk/services/additional/wap.html

[13] Amazon Web Bookshop : http://www.amazon.co.uk/

[14] Ftpmail Service : http://sunsite.org.uk/packages/ftpmail/

[15] Genie Email to SMS Gateway : http://www.genie.co.uk/

[16] Beej's Guide to Network Programming - Using Internet Sockets :
http://www.ecst.csuchico.edu/~beej/guide/net/

[17] The LinuxThreads Library : http://pauillac.inria.fr/~xleroy/linuxthreads/

[18] The Perl Language : http://www.perl.com/

[19] Department of Computing central web server : http://www.doc.ic.ac.uk/

[20] Apache Web Server : http://www.apache.org/

[21] Introduction To Unix Signals Programming :
http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html

[22] The Independent WAP/WML FAQ : http://wap.colorline.no/wap-faq/

[23] Nokia 7110 WAP compliant mobile telephone :
http://www.nokia.com/phones/7110/

[24] Kannel WAP Gateway Project : http://www.kannel.org/

[25] Procmail Mail Filter : http://www.procmail.org/

[26] Sendmail Mail Delivery System : http://www.sendmail.org/

[27] PIKA Inline-GTi Telephony Card :
http://www.pika.ca/Application_Cards/InLineGT.htm

[28] Festival Text-to-Speech Synthesis System :
http://www.cstr.ed.ac.uk/projects/festival/

[29] CVoiceControl Speech Recognition Package :
http://www.kiecza.de/daniel/linux/

[30] Magicpoint Presentation Package : http://www.mew.org/mgp/

[31] Bayonne Telephony Server : http://Bayonne.sourceforge.net/

[32] Ncurses Linux Curses Library :
http://www.informatik.uni-hamburg.de/RZ/software/ncurses/

Appendix A : Software & Hardware Used

This is a list of the main pieces of software and hardware used in this project :

Apache Web server : http://www.apache.org/

Bayonne (Previously named ACS) Telephony server :
http://Bayonne.sourceforge.net/

EGCS C compiler system : http://gcc.gnu.org/

Festival Text-To-Speech synthesis system :
http://www.cstr.ed.ac.uk/projects/festival/

GNU Tools UNIX utilities : http://www.gnu.org/

Kannel WAP gateway : http://www.kannel.org/

Linux UNIX Based operating system : http://www.linux.org/

Lyx Typesetting package : http://www.lyx.org/

Magicpoint X11 based presentation tool : http://www.mew.org/mgp/

Ncurses Linux Curses implementation :
http://www.informatik.uni-hamburg.de/RZ/software/ncurses/

Perl Scripting language : http://www.perl.com/

Procmail Email filter : http://www.procmail.org/

PIKA Inline GTi Telephony card and C library : http://www.pika.ca/

Pthreads Linux POSIX threads implementation :
http://pauillac.inria.fr/~xleroy/linuxthreads/

Redhat Linux GNU/Linux distribution : http://www.redhat.com/

RCS Revision Control System :
http://uk.rpmfind.net/redhat/6.1/i386/rcs-5.7-10.i386.html

Sendmail Mail delivery system : http://www.sendmail.org/

Vim Text editor : http://www.vim.org/

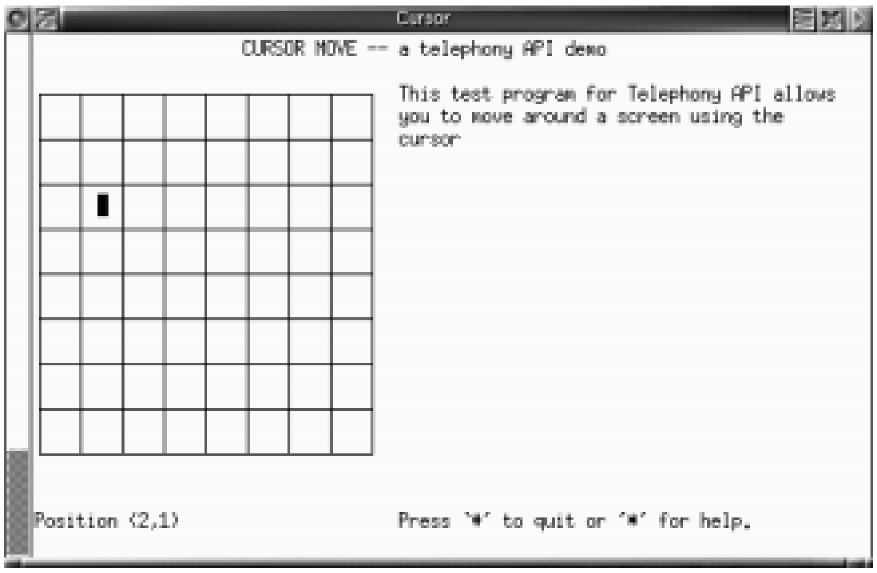
Wml-tools Set of utilities for using wml : http://pwot.co.uk/wml/

Appendix B : Example Code Use of the API

The following code gives an example use of the telephony API by an application. The application in question is named “Cursor” and allows the user to move a cursor around a box in a window on the screen, for example by using the keys of a mobile telephone. A full description of the application is given in section 3.3.1. The application uses the ncurses library to perform the windowing operations. This code shows the simplicity of using the telephony API, and the affect that it has on a program.

The application provides feedback to the user as to their position through the telephony API and allows the user to move the cursor around the grid with input from the API. For the example of access by a mobile telephone the feedback is given as speech. The user’s position is also relayed visually on the screen. This example therefore demonstrates the full power of the abstraction that the API brings. For an example of an inclusion of the library into a more complex application, please see section 3.6.1 which describes the addition of the API into an existing presentation package.

An example usage of the cursor application can be seen in Figure 8.



```
CURSOR MOVE -- a telephony API demo

This test program for Telephony API allows
you to move around a screen using the
cursor

Position (2,1)                 Press '*' to quit or '#' for help.
```

Figure 8: Screen Capture of Cursor Application in use with the API

In the case of this screen capture, the user was accessing the cursor application by telephone using the DTMF interface. In order to initiate the connection, the user dialled an access number. Once connected the application responded with “Welcome to cursor, initial position (0,0)” (as can be seen in the following code) through the API. This was converted to speech by the API and the user heard the response through the ear piece of their telephone. In order to move the cursor to the shown position of (2,1) the user provided two down commands and one right command by pressing the keys on the telephone. After each key press (input to the application) the application responded by moving the cursor and with output of the new position to the user. This was processed by the API so as to be heard by the user even though the application does not have any knowledge of the user's access method. A full discussion of this process for a telephone is given in section 3.4.5.

It is worth noting here that the response is generated entirely at run-time, the synthesised speech is not pre-recorded. A visual response is received almost instantaneously by users accessing the application. If the application programmer had so desired the response could have been entirely visual with no text-to-speech synthesis or it could have been entirely by output to the API with no visual response other than the movement of the cursor. This may well be the case with a graphical application where the programmer does not wish to pollute the display with text. Also, it improves the response time as no output text has to be synthesised in order to be returned to the user.

To compile the code given in this section with the telephony API library it is necessary to link the library at compilation time. With the gcc compiler the necessary flags are -L the path to an extra library and -l for the name of the extra library. Note that the ncurses library[32] is also necessary for this application. An example compilation line with gcc is :

```
gcc -o cursor cursor.c -O2 -pipe -fomit-frame-pointer -Wall
    -lncurses -L/homes/ncet/project/app -ltele_api_lib
```

Note that libtele_api_lib is dynamically linked into the application at run-time as discussed in section 3.5 of this report.

The following is the code for the application, commented to show the use of the telephony API :

```
/*
 * Cursor application to demonstrate telephony API
 */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>

#include "tele_api_lib.h"
```

The header file containing the definitions of the functions in the telephony API is included.

```
/* board size */
#define BDEPTH 8
#define BWIDTH 8

/* where to start the instructions */
#define INSTRY 2
#define INSTRX 35

/* corner of board */
#define BOARDY 2
#define BOARDX 0

/* notification line */
#define NOTIFYY 21
#define CX(x) (2 + 4 * (x))
#define CY(y) (1 + 2 * (y))
#define cellmove(y, x) wmove(boardwin, CY(y), CX(x))
#define CXINV(x) (((x) - 1) / 4)
#define CYINV(y) (((y) - 2) / 2)

static int rw,col;  /* current row and column */
static int lastrow,lastcol;  /* last location visited */
static WINDOW *boardwin;  /* the board window */
static WINDOW *helpwin;  /* the help window */
static WINDOW *msgwin;  /* the message window */

static void init_win(void);
static void play(void);
static void dosquares(void);
```

The includes are followed by a number of definitions which ease the use of the ncurses system.

```
int main(int argc ,char *argv[] )
{
    if (tele_api_init("Welcome to cursor, initial position
                                     (0,0)") < 0) exit(1);
    init_win();
    play();
    if (tele_api_close("Hope you had fun in cursor") < 0)
                                                exit(1);
    endwin();
    return(0);
}
```

The main() function contains two calls to the telephony API. On startup the application calls tele_api_init() with a welcome message for the user. After the user has finished using the application (i.e. after play() has returned) tele_api_close() is called with a final message for the user.

```
static void init_win(void)
{
    initscr ();
    cbreak ();  /* immediate char return */
    noecho ();  /* no immediate echo */
    boardwin = newwin(BDEPTH * 2 + 1, BWIDTH * 4 + 1, BOARDY,
                                                    BOARDX);
    helpwin = newwin(0, 0, INSTRY, INSTRX);
    msgwin = newwin(1, INSTRX-1, NOTIFYY, 0);
    scrollok(msgwin, TRUE);
    keypad(boardwin, TRUE);

    if (has_colors())
    {
        int bg = COLOR_BLACK;

        start_color();
        if (use_default_colors() == OK)
        bg = -1;
    }
}
```

The init_win() procedure simply sets up the “windows” in the ncurses system.

```
static void help(void)
/* game explanation -- initial help screen */
{
    (void)waddstr(helpwin, "\n\nThis test program for Telephony
                                                 API allows\n");
    (void)waddstr(helpwin, "you to move around a screen using
                                                    the\n");
    (void)waddstr(helpwin, "cursor\n");

    (void)mvwaddstr(helpwin, NOTIFYY-INSTRY, 0, "Press ‘#’ to
                                     quit or ’*’ for help.");
}
```

The help screen is a subsection of the display which provides instructions for the user.

```
static void play (void)
/* play the game */
{
    int c, ny = 0, nx = 0, loop=0;
    int help_msg=0;

    /* clear screen and draw board */
    werase(boardwin);
    werase(helpwin);
    werase(msgwin);
    dosquares();
    help();
    wnoutrefresh(stdscr);
    wnoutrefresh(helpwin);
    wnoutrefresh(msgwin);
    wnoutrefresh(boardwin);
    doupdate();
```

The play() function contains the central execution loop of the application, firstly setting up the screens in ncurses.

```
    for (;;) {
        char* msg= (char *) malloc(MSG_LENGTH);
        char* buf= (char *) malloc(MSG_LENGTH);
        if (tele_api_clear_mem(msg, MSG_LENGTH) < 0) exit(1);
        if (tele_api_clear_mem(buf, MSG_LENGTH) < 0) exit(1);
        if (rw != lastrow || col != lastcol) {
            if (lastrow >= 0 && lastcol >= 0) {
                cellmove(lastrow, lastcol);
            }

            cellmove(rw, col);
            lastrow = rw;
            lastcol= col;
        }
        cellmove(rw, col);
        cellmove(rw, col);
        sprintf(msg, "Position (%d,%d)", rw, col);
```

```
werase(msgwin);
waddstr(msgwin, msg);
wrefresh(msgwin);
wrefresh(boardwin);
```

A do-forever loop is used which sets up the windows with the cursor in position. Two buffers are cleared to store the messages that go to and are received from the telephony API.

```
if (help_msg) {
    if (tele_api_clear_mem(msg, MSG_LENGTH) < 0)
                                       exit(1);
    sprintf(msg, "Use the cursor keys to move around
           the screen, or # to quit. The application
                 will respond with your position.");
    help_msg=0;
}
if (loop!=0) {
    if (tele_api_output(msg) < 0) exit(1);
}
```

In every iteration of the loop apart from the first tele_api_output() is called with a message for the user. This message is either the position of the cursor, or a help message if this was requested.

```
free(msg);
if (loop==0) loop++;

if (tele_api_input(buf) < 0) exit(1);
```

The application then waits for input from the user of the telephony API by virtue of a call to tele_api_input(). Once this is received the application can act upon this input.

```
c=buf[0];
free(buf);
switch (c) {
    case 'k': case '8': case 8: case KEY_UP:
        ny = rw+BDEPTH-1; nx = col;
        break;
    case 'j': case '2': case 2: case KEY_DOWN:
        ny = rw+1; nx = col;
```

```
                break;
            case 'h': case '4': case 4: case KEY_LEFT:
                ny = rw; nx = col+BWIDTH-1;
                break;
            case 'l': case '6': case 6: case KEY_RIGHT:
                ny = rw; nx = col+1;
                break;
            case 'y': case '7': case 7: case KEY_A1:
                ny = rw+BDEPTH-1; nx = col+BWIDTH-1;
                break;
            case 'b': case '1': case 1: case KEY_C1:
                ny = rw+1; nx = col+BWIDTH-1;
                break;
            case 'u': case '9': case 9: case KEY_A3:
                ny = rw+BDEPTH-1; nx = col+1;
                break;
            case 'n': case '3': case 3: case KEY_C3:
                ny = rw+1; nx = col+1;
                break;
            case '*':
                help_msg=1;
                break;
            case 'q': case '#':
                goto getout;
                break;
            default:
                break;
        } // switch
```

A simple case statement is used to act on the input of the user. Note that if the telephony API standard calls for a help message (*) or to quit (#) are given by the user then they are adhered to by the application.

```
        col = nx % BWIDTH;
        rw = ny % BDEPTH;
    } //for

    getout:
}

static void dosquares (void)
{
    int i, j;
```

```
mvaddstr(0, 20, "CURSOR MOVE -- a telephony API demo");
```

The dosquares() function follows a simple set of ncurses calls to draw the grid around which the cursor can move.

```
move(BOARDY,BOARDX);
waddch(boardwin, ACS_ULCORNER);
for (j = 0; j < 7; j++) {
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_TTEE);
}
waddch(boardwin, ACS_HLINE);
waddch(boardwin, ACS_HLINE);
waddch(boardwin, ACS_HLINE);
waddch(boardwin, ACS_URCORNER);

for (i = 1; i < BDEPTH; i++) {
    move(BOARDY + i * 2 - 1, BOARDX);
    waddch(boardwin, ACS_VLINE);
    for (j = 0; j < BWIDTH; j++) {
        waddch(boardwin, ' ');
        waddch(boardwin, ' ');
        waddch(boardwin, ' ');
        waddch(boardwin, ACS_VLINE);
    }
    move(BOARDY + i * 2, BOARDX);
    waddch(boardwin, ACS_LTEE);
    for (j = 0; j < BWIDTH - 1; j++) {
        waddch(boardwin, ACS_HLINE);
        waddch(boardwin, ACS_HLINE);
        waddch(boardwin, ACS_HLINE);
        waddch(boardwin, ACS_PLUS);
    }
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_RTEE);
}

move(BOARDY + i * 2 - 1, BOARDX);
```

```
    waddch(boardwin, ACS_VLINE);
    for (j = 0; j < BWIDTH; j++) {
        waddch(boardwin, ' ');
        waddch(boardwin, ' ');
        waddch(boardwin, ' ');
        waddch(boardwin, ACS_VLINE);
    }

    move(BOARDY + i * 2, BOARDX);
    waddch(boardwin, ACS_LLCORNER);
    for (j = 0; j < BWIDTH - 1; j++) {
        waddch(boardwin, ACS_HLINE);
        waddch(boardwin, ACS_HLINE);
        waddch(boardwin, ACS_HLINE);
        waddch(boardwin, ACS_BTEE);
    }
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_HLINE);
    waddch(boardwin, ACS_LRCORNER);
}

/* cursor.c ends here */
```

Appendix C : Obtaining The API

The deliverables produced during the project are available for download. They available products can be found at :

`http://www.doc.ic.ac.uk/~ncet/project/`

At this location the following items are available for download :

- The source code of the implementation of the API as a library (C)
- The source code of the service daemon implementation (C)
- The source code of the implementations for the device access methods (C and Perl)
- The source code of the cursor application (C)
- The source code of the magicpoint application (C)
- The original project proposal (html and postscript formats)
- The final version of this report (postscript format)
- The slides for the project presentation (magicpoint and postscript formats)