

SIP / H.323 / AIX telephone customization

Submitted in partial fulfilment
of the requirements of the degree
Bachelour of Science Honours in Computer Science
at Rhodes University

Brendan Ronald Marlborough

Supervised By:

Professor Alfredo Terzoli

Professor Peter Clayton

November 2005

Acknowledgements

I would like to thank Alfredo Terzoli and Peter Clayton for supervising my project and providing guidance throughout the year. I would also like to thank Fred and Justin for helping with Asterisk and iLanga, and Hannah for proof reading my paper. Lastly

I would like to thank the Department of Computer Science and all my project sponsors – Telkom, Comverse, Verso Technologies, and Thrip – for making this project possible.

Abstract

VoIP has become a cheap alternative to using the traditional phone network. Many people however don't have access to a computer and require hardware VoIP phones. The Palm 1 is a cheap VoIP phone that is currently being used at Rhodes University. The phone does however have a few negative features, which can be improved. It comes with an API which allows it to be reprogrammed and all of its source code is available to the public. This report describes the hardware and software modules that form the phone, and how this enables the phone to be customised in order to improve some of its features.

Contents

CHAPTER 1: INTRODUCTION.....	5
1.1. INTRODUCTION	5
1.2. VOIP HISTORY	6
1.3. COMPARISON OF COST AND FUNCTIONALITY.....	7
1.4. ILANGA AND ASTERISK.....	8
1.5. PROBLEM AND SOLUTION	9
CHAPTER 2: PA1688 HARDWARE	10
2.1 CHIP STRUCTURE	10
2.2 CHIP ARCHITECTURE	12
2.2.1 Controller	12
2.2.2 DSP	13
2.2.4 SDRAM Interface	14
2.2.5 AC97 Codec.....	14
2.2.6 Keyboard interface.....	14
2.2.7 Network interface.....	15
2.3. PALM1 HARDWARE	15
2.4. BANK SWITCHING	16
CHAPTER 3: SOFTWARE.....	17
3.1. DATA STRUCTURE	17
3.2. DMA BUFFER STRUCTURE	19
3.3. PROGRAM STRUCTURE.....	21
3.3.1. Languages and Code Structure	21
3.3.2. Program Flow.....	22
3.4. API LIBRARIES	24
3.5. DISPLAY.....	25
CHAPTER4: DEVELOPMENT ENVIRONMENT AND COMPILING.....	27
4.1. COMPILER AND ENVIRONMENT	27
4.2. OTHER COMPILERS	28
4.3. COMPILING	28
4.4. UPDATING FIRMWARE.....	29
4.5. UPGRADE ISSUES	30
4.6. FAILURE RECOVERY	31
CHAPTER 5: CHANGES IMPLEMENTED	33
5.1. MODIFYING THE DISPLAY AND SHOWING THE TIME	33
5.1.1. Modifying message strings	33
5.1.2. UI update handler.....	34
5.1.3. Displaying the time	35

5.2.	MISSED CALLS	37
5.2.1.	Storing miss call data	37
5.2.2.	Showing if there is a missed call.....	38
5.3.	MUTE BUTTON.....	40
5.3.1.	Key Event Handler.....	40
5.3.2.	Muting the input Volume.....	41
5.4.	HIDING THE ILANGA PASSWORD.....	42
5.5.	FEATURES NOT IMPLEMENTED	44
5.6.	IMPLEMENTATION CHALLENGES	45
CHAPTER 6: CONCLUSION AND FUTURE WORK		47
6.1.	CONCLUSION.....	47
6.2.	FUTURE WORK	47
6.2.1.	Phone Security	48
6.2.2.	Encrypting the media stream	48
REFERENCES.....		49

Chapter 1

Introduction

This chapter introduces Voice over IP (VoIP) and gives a history of the trend in telecommunications. The need for a cost effective VoIP end device will be highlighted, and a comparison between the cost and functionality of VoIP devices will be discussed. Finally, the problem statement for this project and the solution will be explained.

1.1. Introduction

The phone has been the preferred way to communicate since the 1900's. The phone network grew and now almost everyone in the developed world has access to a handset. Until recently the phone network was the largest communication network in existence, providing voice and fax services to billions. Unlike the phone network, the internet is a public network and connection to it is cheap, costing a local call at the most. When the internet started to grow and become more popular, email was soon adopted and could be used to send documents anywhere in the world for free. As technology grew and internet connection speeds started increasing, people started to make voice calls using this vast network.

Organizations with network infrastructures in place could also start replacing their private branch exchange systems with cheaper VoIP systems. Calls to offices in different geographical locations could also be routed over an existing data link rather than having to pay for a costly national or international call.

It is now possible to call someone anywhere in the world for only the price of the connection to the internet. The problem with this is that people like to use things that they know and understand. Everyone knows how to use a phone, but you cannot use a

normal phone to make calls over the internet. Because of this, people started developing techniques to make people more comfortable with using VoIP, by making the interface as much like a traditional phone as possible.

1.2. VoIP history

The first way to communicate by VoIP was by running software on a PC. Vocaltec, Inc. released the first internet phone software in 1995 [5], with software that was designed to run on a 33MHz 486 PC. There are still many software phones which allow you to communicate with VoIP using a PC. These software phones essentially turn the PC into a phone by providing similar interfaces of a conventional phone. The PC requires speakers, a microphone and an on-screen button display for dialling. These software phones work well, however a computer is needed to perform the encoding and decoding of audio and sending and receiving of network packets. The user is also required to be computer literate and many of today's phone users don't necessarily know how to use a computer.

Using phone adapters [7] is another way to enable VoIP services. These devices allow you to convert a conventional phone into an IP phone. These devices typically have two interfaces: an Ethernet interface and an analogue phone interface. These devices allow you to have more than one phone device on a single Ethernet link. By using a phone adapter, a phone and a fax machine can be operated on the same Ethernet link with different numbers, essentially mimicking two different phone lines. These devices allow users to keep their old phones, while making it possible to call on an IP network.

The use of phone adapters is a way to enable VoIP with traditional phones, however one can not use adapters forever and eventually traditional phones will stop being produced. This has caused developers to start manufacturing phones specifically for VoIP. These phones plug directly into an Ethernet port and have a built in processor for handling Ethernet and IP packets. These phones come in a variety of form factors, feature sets and user interfaces [14]. The primary distinction between phone forms is if they are designed for desktop or mobile use. Desktop phones are typically used on

an office desktop and usually have a 2 line LCD display. Mobile phones are smaller and are portable, some may have a base station that connects to the network while others use an 802.11 network for connectivity.

Rhodes University is currently using hardware VoIP phones on its network. One of these phones is the Palm1 from Centrality Communications. This phone is cheap compared to other hardware VoIP phones, but does lack some of the functionality of more expensive VoIP phones. The next section describes the trade-off between cost and functionality in VoIP phones.

1.3. Comparison of cost and functionality

VoIP phones are becoming very popular and there are a wide variety of them available. As with most modern devices there is a trade-off between the cost and functionality of these phones. Two phones will be discussed with regard to their cost and functionality: the Palm1 from Centrality Communications, [2] and the IP10S from Swiss Voice [13].

The Palm1 and Swiss Voice are desktop phones that have 2 Ethernet connections (this allows the phone to act as a switch and lets you plug in a PC to the phones freeing up an Ethernet port). They also support the common VoIP protocols: SIP, H.323, MGCP; as well as common IP protocols such as DHCP. Most of the low level features of the phones are very similar, as well as features such as call lists, number lists and speed dialling.

One thing that is quickly noticeable is the overall quality of the phones. The Palm1 is bulkier and the plastic used is of lower quality than that of the Swiss Voice phone. The display of the Swiss Voice is also better quality having a 128x64 pixel full graphic display rather than a two line array of 8x5 character blocks as in the case of the Palm1. The Swiss Voice phone also supports power over Ethernet while the Palm1 must be plugged in to an AC outlet. This is important, as in traditional telephony a simple phone does not need to be plugged in to mains power and can operate off the voltage supplied through the phone cable. The Palm1 also has some

annoying features such as constantly displaying the firmware version on the display, and flashing an LED whenever there is network traffic.

Some IP phones go one step further from being just a replacement for a traditional phone. Cisco Systems range of phones for example can support additional information services including XML capabilities. XML-based services can be customized to provide users with access to a diverse array of information such as stock quotes, employee extension numbers, or any Web-based content [3]. These phones are primarily for users who want top of the range equipment. Most customers who can afford Cisco Systems phones will have computers that can provide similar functionality anyway. As mentioned before, the Swiss Voice is a better quality phone which costs around R800. The Palm1 is cheaper at around R450, and performs the same functions as the Swiss Voice.

1.4. iLanga and Asterisk

iLanga is a computer based PBX which was built in the Computer Science department at Rhodes University. The main component of iLanga is Asterisk. Asterisk is an open source, converged time division multiplexing (TDM) and packet based communication system [9]. iLanga also consists of two other components, they are: SIP express router and Open H323 gatekeeper [9].

The Palm1 is used as an end device to iLanga within the Computer Science department at Rhodes University, and was recently provided to students in residence. Because the Palm1 is commonly used as an end device to iLanga, it would be preferable to customise some of the features of the phone, so that it works better in the iLanga environment.

The use of iLanga is free, however it does have connections to the external phone network. Dialling to an outside line is billed for and requires the authentication of the user. The user will be asked to enter their details while on the phone, the problem with this is that the phone displays all the characters when they are pressed on the keypad,

therefore the phone will also display the password on the display. A way to conceal the password would be a good feature to be implemented.

Asterisk also provides voicemail accounts. When a user receives a voicemail message, the user is notified by email. It would also be nice for the user to be notified on the display of the phone. This would be made possible by a SIP notify message, which provides support for message waiting indicator messages [8].

1.5. Problem and Solution

The decrease in cost to connect to the internet has caused many people to start using VoIP. The majority of people however do not have access to a computer and a cheaper way of gaining access to VoIP is needed. Hardware VoIP phones are still a new technology and are therefore quite expensive. A cheap VoIP phone has been found and is being used at Rhodes University. The phone has a few negative features and could be integrated better with iLanga. The Palm1 comes with an API which allows it to be re-programmed, and all its source code has been released to the general public. Because of this, the phones negative features can be improved to make it a cost effective solution to providing access to VoIP.

In order to fix the negative features of the phone, the hardware and software of the phone will need to be understood. A good understanding of the API functions will also need to be gained, as well as the tools used to compile the code and update the phones firmware. The improvements will have to be implemented in the code, and the phone will need to be tested to see whether the changes were successful. The remainder of this report will address these negative features, along with a detailed description of the steps I have taken to rectify the problems.

Chapter 2

PA1688 Hardware

This chapter gives an explanation of the hardware used in the Palm1 IP phone. The structure of the PA1688 chip will be discussed, as well as its architecture. The peripheral hardware will also be discussed in some detail. Most of the content from this chapter was obtained from the phones development guide [1].

2.1 Chip Structure

The Palm1 phone is powered by a PA1688 processor which is manufactured by Centrality Communications. The PA1688 is a single chip which consists of a controller, dual processor, digital signal processor and other interfaces for SDRAM, flash memory, and AC97 audio chips. By supporting common peripheral interface chips which are well supplied and low in cost, the overall manufacturing cost of the phone is reduced. The chips small die and software size make the chip itself a low cost solution for phone manufacturers. It also allows customisation for different systems with different customer requirements [10]. Figure 1 below gives an overview of the structure of the PA1688 chip, as well as the buses linking each part of the chip.

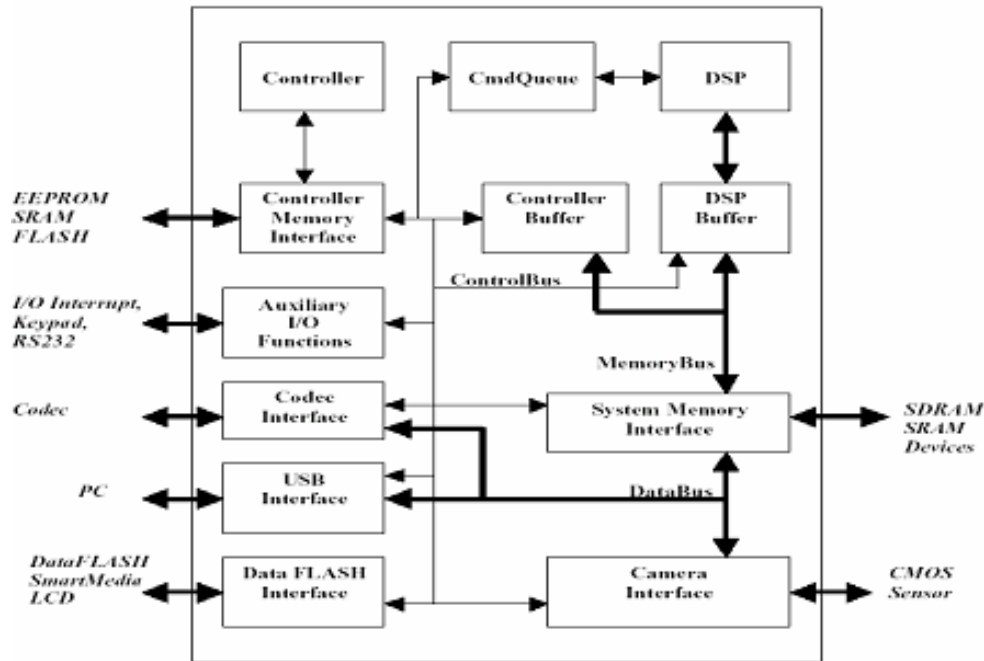


Figure 1: Structure of the PA1688 chip [1]

The core of PA1688 chip consists of two parts.

- Controller Enhanced Intel MCS8051 instruction compatible controller. This is used to control general purpose tasks such as dealing with key presses and updating the display. It also handles the Ethernet protocols such as TCP/IP and DHCP and the VoIP protocols such as H.323 and SIP. The operation rate of the PA1688 chip is 50 MHz, taking between 4 and 8 clock cycles per instruction. This is equivalent to a standard MCS8051 processor operating at 100MHz due to the integrated controller.
- An ADSP2181 instruction compatible digital signal processor. The main purpose of the DSP is to compress the raw digital audio from the AC97 chip into one of the VoIP audio formats, as well as decompress from these formats into uncompressed audio. The phone supports the following audio codecs: g729, g7231, g711u, g711a, GSM, iLBC. The DSP runs at 33MHz, equivalent to the standard 2181 DSP. By having a dedicated chip to handle compression / decompression of audio, the main processor can be much less powerful while still being able to perform general purpose tasks.

Apart from the two core components mentioned above, the PA1688 chip also has interfaces for RS323, USB, SDRAM, AC97codec, keypad and LCD display. RS232 and USB are not used in the phone at present.

2.2 Chip Architecture

This section describes the architecture of the chip, including the memory architecture of the controller and its peripheral interfaces such as SDRAM and flash memory. An understanding of the memory architecture is required for Chapter 3.

2.2.1 Controller

The controller core consists of the main controller as well as the CPU. The CPU is an 8 bit micro-controller and is completely compatible with the Intel 8051 instruction set.

The controller has an internal cache memory of 256 bytes and 64Kb of external data memory. Of the 64Kb of external memory 4.5 Kb can be used for program data, the remainder is allocated for internal registers, DSP memory allocations, and memory exchange buffers. As can be seen in figure 2, the 4.5Kb of program data memory consists of a 512 byte data exchange buffer which is used to transfer data to and from the larger amount of SDRAM. The speed at which these various types of memory can be accessed differs greatly, with internal cache being the fastest and SDRAM being the slowest. The variable types that can be used to access the different memory spaces will be discussed in Chapter 3.

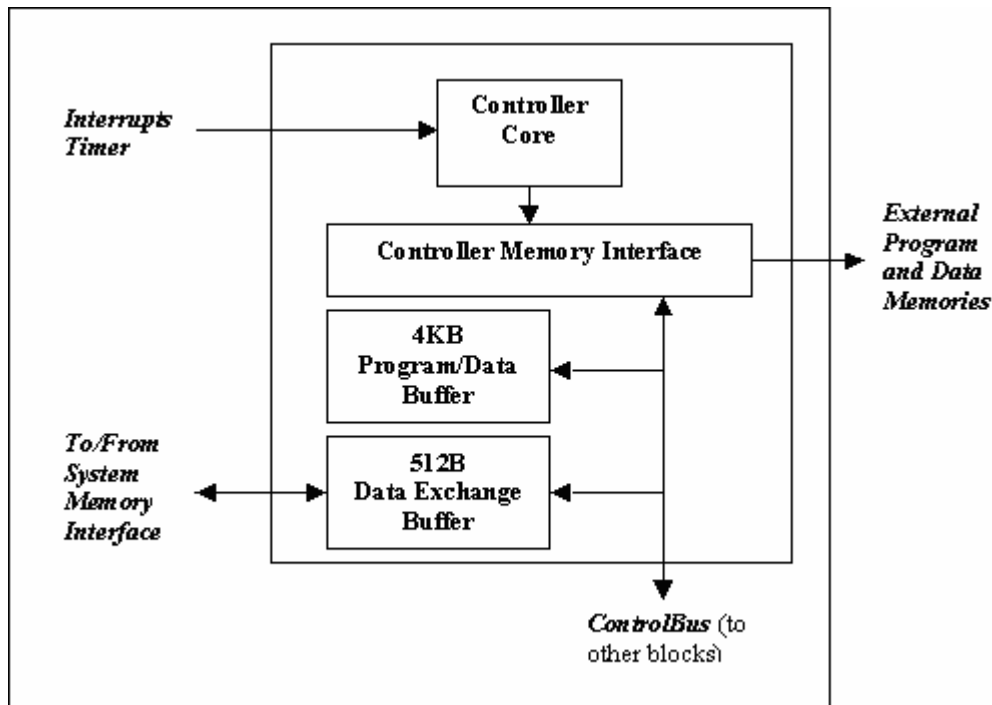


Figure 2: Controller Memory Structure [1]

2.2.2 DSP

The DSP is an ADSP2181 compatible instruction set DSP and is used for specific processing tasks such as encoding and decoding audio. The controller manages the execution of the DSP and transfers data to and from the DSP through the 512 byte exchange buffer located in the controller core. None of the code written for the DSP was studied or modified during this project and no further investigation into the operation of the DSP was needed.

2.2.3 Program Flash Interface

The PA1688 has an integrated interface for flash memory. The interface supports 8 bit data line width flash memory up to 16MB. Only the first 8MB can be used for programming content. The flash memory contains all the program memory as well as initializing data. The main controller fetches its instructions from the program flash while running. It therefore has a similar function to read only memory (ROM), however unlike with ROM, flash memory can be read and erased one sector at a time.

2.2.4 SDRAM Interface

The PA1688 supports up to 16MB of SDRAM. The SDRAM is used for temporary data such as AC97 and DSP buffers as well as other data such as missed, answered and dialled calls. When the phone boots, all TCP/IP address and routing information is loaded from the flash memory into SDRAM. The content of the SDRAM is separated into blocks with some blocks reserved for programming use.

Because the MCS8051 is an 8-bit controller it can only address up to 256 bytes. In order to address the entire 16MB of SDRAM, 3 pointers are needed, giving an effective 24 bit pointer. The 16 MB can therefore be regarded as $256*256*256$. The SDRAM is therefore broken up into 65536 blocks of 256 bytes, and the pointers consist of a high-byte index, a low-byte index and a length pointer. The use of these pointers will be explained further in Chapter 3.

2.2.5 AC97 Codec

The AC97 chip is used to convert analogue audio to digital audio and vice versa. The phone has two microphones and two speakers, one pair for the handset and one pair for when the phone is in speaker / hands free mode. The sound signal is decoded/encoded at 8 KHz which is the sampling rate used in most VoIP codec's.

When the phone is in operation the controller, copies the uncompressed digital audio from the AC97 codec into the SDRAM voice buffer. The DSP then gets the uncompressed audio from the buffer, encodes it to the selected VoIP codec and puts it back into SDRAM. The controller then adds the RTP / TCP/IP headers and sends the packet over the network. The reverse occurs when a packet is received from the network, which eventually gets played as analogue sound to one of the speakers.

2.2.6 Keyboard interface

The CMOS Logic chip used in the PA1688 for the keypad input has several pins which create a 3x4 array. This is enough for a traditional phone that only has 12

buttons, however it is not enough for the IP phone which has many other functions and the amount of pins are expanded using programmable IO to create a 4x7 array.

2.2.7 Network interface

The network interface is a NE2000 10MB interface which is used in most 10M network cards.

2.3. Palm1 hardware

There are different variations of the PA1688, specifically regarding the type and amount of Flash Memory and SDRAM. Although there are a large number of phones with different designs with regard to button layout and casing, there are only a few different hardware designs. The Palm1 is based on a variation of the PA1688 known as the PA168s. Figure 3 shows a picture of the Palm1 phone.

The PA168s contains the PA1688 chip and uses 1MB of AM29LV008T memory from AMD. It also has 2MB of SDRAM.

The way the code is organised in the flash memory depends on what flash memory is being used. Each different type contains a different number of sectors of different sizes. The AM29LV008T memory used in the PA168s design contains fifteen 64Kb sectors, a 32Kb sector, a 16Kb sector and two 8Kb sectors. Because writing and reading from smaller sectors is much faster all the phones settings are stored in the first of the 8Kb sectors. This enables the phone to update its settings much faster than if the settings were in a larger sector.



Figure 3: Palm1 IP phone

2.4. Bank Switching

As mentioned previously the controller only has 64 KB of external memory and can only address up to 64 KB. This is not enough for all the code needed to control the phone and so bank switching is used.

Bank switching allows the controller to access more memory than it can actually address. The PA168s has 1MB of flash memory however the controller can only address 64 KB. With bank switching, the code is split up into different segments and during execution the controller switches between these segments, allowing the code size to be much bigger [6]. The flash memory used is split up into 64 KB sectors, allowing 1 segment of code to fit into 1 sector of memory. The flash memory used in the PA168s has fifteen 64 KB sectors and the size of the final binary code file is 960 KB (15*64KB) when compiled.

The way the code is split into the different banks as well as how to switch between banks is discussed in Chapter 3.

Chapter 3

Software

All the code for the phone has been released to the open source community, with the recent addition of all the phones libraries. This chapter describes the data structures of the PA1688 chip and how they can be used in code. The structure of the DMA buffer, which is used to access the SDRAM, is also discussed. Finally the structure and flow of the code which controls the phone is explained.

3.1. Data Structure

The MCS8051 micro-controller has several data buffers, these buffers are controlled with different variable types. By using the different variable types correctly, one can improve the efficiency of the program. The different variable types that correspond to each memory type will now be described separately.

Figures 4 and 5 show a graphical representation of the allocation of these data types, highlighting the size and position of each data structure.

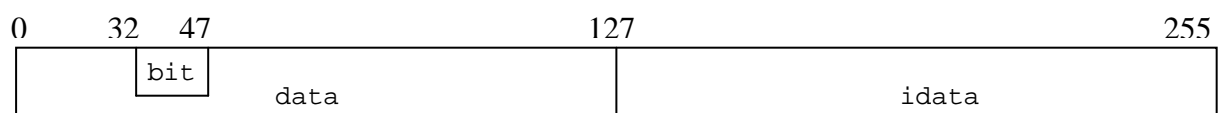


Figure 4: Internal cache memory allocation

- **data**

This is located in the internal cache of the controller. Its address range is from 0 – 127 bytes. Because the data is located in the internal cache it can be written to and

read from faster than the other variable types, however there is little space available.

- **idata**

This is also located in the internal cache however it is used for the program stack. Its address space is from 128 – 255 bytes and the stack pointer moves from the end of the address space to the beginning.

- **bit**

The `bit` type is also located in the internal cache in the 0x20 - 0x2f address space. The read / write speed is the same as `data` and `idata` but every variable uses only one bit. The length of this field is 16 bytes or 128 bits allowing for 128 variables.

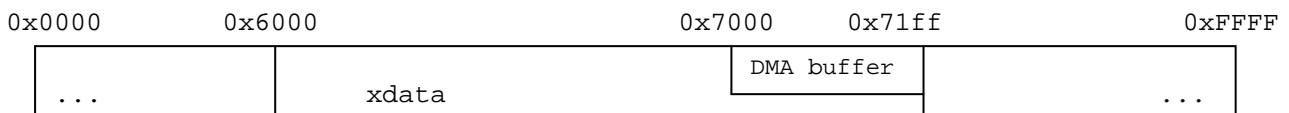


Figure 5: Controller SRAM memory allocation

- **xdata**

This data type is located in the built in 4.5 KB of SRAM in the controller. The address range is from 0x6000 to 0x71FF. Because this data is located outside the controller core it is slower than `data` and `idata`. The 4.5 KB of SDRAM is divided into two parts. The first 4 KB can be addressed by the `xdata` type. The remaining 512 bytes can also be addressed by the `xdata` type but is also used as a DMA buffer used to access the SDRAM.

- **code**

The `code` type is located in the flash memory outside the controller and is 64 KB in size. Because the flash memory is located outside the controller on a separate interface it is the slowest variable type to access.

The data fields described above correspond to the following data types which can be used in the code for the phone:

- `DCHAR`, `DSHORT`, `DLONG`: Used to define char, short and long variables in the `data` field.
- `BOOLEAN`: Used to define Boolean variable in `bit` field.
- `XCHAR`, `XSHORT`, `XLONG`: Used to define char, short and long variables in the `xdata` field.
- `CCHAR`, `CSHORT`, `CLONG`: Used to define char, short and long variables in the `code` field.
- `PXCHAR`, `PXSHORT`: Pointers to `XCHAR` or `XSHORT`.
- `PCCHAR`, `PCSHORT`: Pointers to `CCHAR` or `CSHORT`.

3.2. DMA buffer structure

The DMA buffer is located in the last 512 bytes of the `xdata` memory space of the controller. The DMA buffer is used whenever the SDRAM is written to or read from.

In the code for the phone there are 4 buffers defined as follows:

```
Sdram_pDataCache      0x7000
Sdram_pDataCache1    0x7080
Sdram_pDataCache2    0x7100
Sdram_pDataCache3    0x7180
```

Each of these buffers is 128 bytes in length giving a total of 512 bytes. With the following two functions and the buffers above, it is possible to read from and write to SDRAM. The two functions are as follows:

```
SdramRead (iIndexHigh, iIndexLow, iLength)
```

```
SdramWrite (iIndexHigh, iIndexLow, iLength)
```

SdramRead will copy the data at the specified address in SDRAM to the beginning of the DMA buffer.

SdramWrite will copy the data in the DMA buffer into the specified address in the SDRAM.

For example if one wanted to read the data in SDRAM at the address 0x041125, it would be done in the following way.

```
iIndexHigh = 0x04
```

```
iIndexLow = 0x11
```

```
iLength = 0x25 = 37, (37 / 4) + 1 = 10
```

iLength is calculated as a DWORD (double word) so it must be divided by four.

The data in the SDRAM can be copied into the DMA buffer using the SdramRead function.

```
SdramRead (0x04, 0x11, 10);
```

The data will now be in the DMA buffer (Sdram_pDataCache) and can be accessed directly. The specific byte of data can be obtained by using the last two hexadecimal digits as the offset.

```
someData = Sdram_pDataCache[0x25];
```

Data can be written to SDRAM by first writing the data into the DMA buffer and then calling the SdramWrite function as described above.

3.3. Program Structure

This section describes the source code which is distributed with the phone. An explanation of the languages used and the overall structure of the code will be given. The basic flow of the code when the phone boots will also be explained.

3.3.1. Languages and Code Structure

All the code for the phone is written in a specialised form of C. As explained above there are different variable types for the different areas of memory space. If one were to declare a character variable in flash memory then the type would be CCHAR, one could not use normal C variable types such as `char` or `int`. The phone also comes with a tool which allows its settings, ring tones, speed dials and other options to be set from a PC. This tool is also used to update the phones firmware over the network. All the source code for this tool is available and is written in C++ using Microsoft's Visual Studio 6 as a development environment.

The code for the phone is distributed in a compressed format. When uncompressed a root directory called PalmH323 is created with various sub-directories in it. The directory structure of the phones code is related to the bank switching style of the processor. Each directory maps to a certain bank of code. For example the code for the `page0` code bank will be located in the directory `\PalmH323\page0`.

The code for the phones update tool is located in the directory `\Palmtool`. All header files are located in the directory `\inc`. The batch files for setting up path information as well as compiling are all located in the root directory. Compiling will be explained in chapter 4.

The code that is provided by the chip manufacturers is for all the different versions of hardware that is used with the different types of phones. The code for each different hardware type must therefore be separated in the code. The way this is done is with

the use of `#ifdef` statements in C. For example if a certain function was only called for PA168s hardware then one would use the following code:

```
#ifdef PA168s
    Function();
#endif
```

This method is used not only for the different hardware types but also for which of the different VoIP protocols are being used as well as which language (English, Chinese, etc.) is being used. These parameters are all specified at compile time.

The header files containing all variable and function declarations are not found in the directories for each bank. They are all located in the directory `\inc`. Any new function that is written should be declared in the appropriate header file. When writing code for the phone, there are many places where memory addresses need to be referenced. Rather than using 'magic' numbers in the code, one should declare constants in the appropriate header files. This allows for easier changes and also makes the code more readable.

3.3.2. Program Flow

The flow diagram as seen in figure 6 shows the different banks that are used as the phone boots up. The diagram shows the program flow when using AM29LV008T flash memory, phones types with different memory will have code banks in different sectors and the flow diagram for these phones will be different.

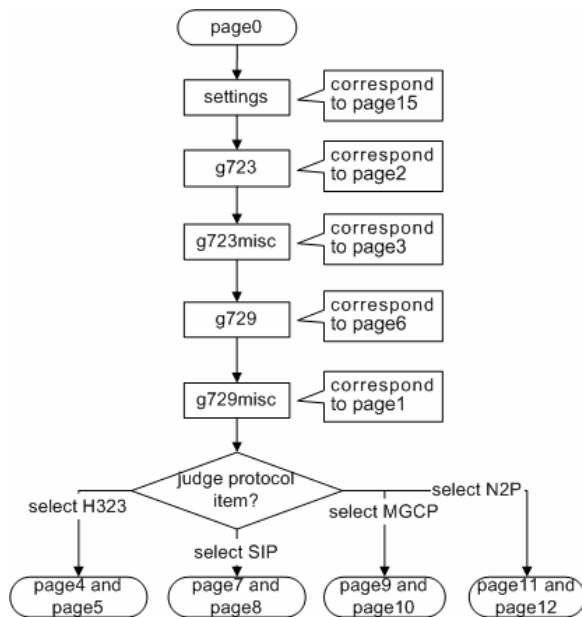


Figure 6: Program flow for AM29LV008T flash memory [1]

The controller starts executing code at page0 as seen in Figure 5. This code is used to perform recovery, testing and read G711 dsp code from flash memory into SDRAM. The program then switches to the next code bank depending on what type of flash memory is used. In the PA168s the flash memory is AM29LV008T so the controller switches to page 15 (the code in the settings directory) as shown in the code below.

```

#if defined AM29LV004T || defined MX29LV004T
    SwitchPage(7);
#elif defined AM29LV008T || defined MX29LV008B || defined MT28F016
    SwitchPage(15);

```

In the settings code bank all the phones settings are loaded from flash memory into SDRAM. The code for controlling the phones menu is also located here. The controller then switches to the g723 and g729 code banks where the DSP is initialised and DTMF and Ring data are loaded into SDRAM.

The controller switches to the next bank depending on which VoIP protocol is being used. The protocols supported are: SIP, H323, MGCP, N2P and IAX2. In earlier versions of the phone, the code was compiled with all protocols and the protocol to be

used could be changed without needing to update the firmware. In later versions however, the code became too large and the protocol to be used needed to be determined at compile time. To change the protocol one needs to recompile the source code and update the firmware.

Once all initialisation has occurred, the code in the `main` bank is run. This code is located in the directory `\main`. Most of the code which deals with the normal operation of the phone is found here. The `main.c` file in this directory contains the main loop which is run when the phone has finished initialising. Here the phone runs through the loop, running tasks such as updating the display and calling the TCP/IP library. Other events such as key presses will generate an interrupt, which then gets handled.

3.4. API Libraries

There are a number of functions in the API libraries which can simplify programming greatly. It hides a large amount of low level manipulation of the PA1688, making it more powerful and easier to use. The main functions of each library are described below. Each library contains one `.lib` file in the `\lib` directory and one `.h` file in the `\inc` directory.

Many of these libraries have been declared open source and the code is provided. The code for the `P_SIP` library can be found in the `\P_SIP` directory for example.

PALM1LIB.

This is the library which is used to control the PA1688. The functions it contains all perform low level manipulation of the controller such as setting control registers. It also optimises assembly to make the system more efficient.

P_DSP.

This library contains all the functions for initializing and controlling the digital signal processor. The aim is to shield the user from the internal workings of the DSP and make it look like it is part of the main controller.

P_TCPIP.

This library contains functions for the lower level network protocols such as TCP/IP, UDP and DHCP. There are many other functions external to this library which enable the use of higher level protocols such as FTP and DNS.

P_RTP.

This library contains the functions for the RTP protocol. RTP is used to transmit the media stream and therefore is used whether SIP, H323 or any other signaling protocol is being used.

P_H323 / P_SIP / P_MGCP.

These libraries contain functions for handling each of the signaling protocols supported by the phone.

P_UTIL.

The P_UTIL library is similar to the standard function library in C. It contains functions for string manipulation and comparison, random number generation and conversion functions.

3.5. Display

The phone's display consists of a 2 line display, with each line displaying 16 characters. Each line of the phones display can therefore be represented as an array of characters. The display buffer is called `_pDisplayCache` and is of type `XCHAR`, which is located in the controllers SRAM.

To simplify writing to the display there are three functions that can be used.

```
DisplayLine(PXCHAR pBuf, UCHAR iLine)
DisplayString(PCCHAR pDisplay, UCHAR iLine)
DisplayVal(USHORT iVal, PCCHAR pPrefix, UCHAR iLine)
```

To display a string of characters the first two functions can be used. The first argument of `DisplayLine` takes a pointer to an array of characters in the `xdata` memory space. `DisplayString` takes a pointer to an array of characters in the code memory space in flash memory. `DisplayString` converts the `PCCHAR` to a `PXCHAR` and calls `DisplayLine`, which saves the user from doing the conversion.

`DisplayVal` takes both a string and a value. It displays the string as a prefix and then displays the value afterwards. This is useful when the value needing to be displayed is not known or can change. For example to show the volume of the phone one would pass the value of the volume level, for example 12, and pass a prefix string of "Volume". The output on the display would be "Volume:12".

All three functions also take an argument `iLine`. This is to indicate which line of the phones display should be written to. The top line of the display is indicated by 0 and the bottom line is indicated by 1. All of these functions also write to the phones display buffer, and therefore the phone will continue displaying the same string until the display buffer is updated to a different value.

These functions are located in the `p_apps` directory in the file `lcd.c`. This file also contains other functions for initialising the display as well as lower level control of the display. Although there is an easy way to display a value with a string as a prefix, there is no function to display a prefixed value with a string after it.

In some cases it is better to have the value prefixed to the string, as in the case of showing something like how any missed calls there are. It would be easier for a user to read 2 missed calls than Missed calls:2. A function has been written that will do this. This is discussed in Chapter 5.

Some other display functions can be found in the file `display.c` also in the directory `p_apps`. These functions have more specific purposes. One such function is `DisplayOutputVol()` which takes no parameters, but gets the volume level and calls the `DisplayVal` function prefixing "volume" before the volume level.

Chapter 4

Development Environment and compiling

This chapter describes the development environment of the project. The compiler used to compile the code is discussed as well as the tools required to update the phones firmware. The steps to compile the code will be explained as well as some issues encountered while compiling, and how to recover the phone should something go wrong. Throughout this chapter, any unreferenced content has been obtained from [1].

4.1. Compiler and environment

There are various tools used to compile the code and update the phone's firmware. To compile the code, we have used the Keil7 compiler. The Keil7 compiler compiles the C code into the 8051 instruction set that is understood by the controller. Only the compiler and linker of the Keil suite are used and not the development environment.

The Keil compiler is meant for commercial use and is therefore very expensive for an individual user. Keil provides a free version of this compiler that only compiles object files up to 2 KB. This is too small for some of the phones object files and the full firmware binary file cannot be produced. Fortunately a test version of the Keil7 compiler was obtained which is able to compile the full sized objects required to make the firmware binary. There are other C to 8051 cross-compilers available and they will be discussed below.

4.2. Other Compilers

Another compiler that was considered was the Small Device C Compiler (SDCC) which is a free, open source C to 8051 compiler [4]. Although this compiler would be able to compile the code for the PA1688 micro-controller, it may not be able to compile the code for the DSP. Another problem is that the developers of the phone have set up all the batch files and version files to work with the Keil7 compiler. One would require a good understanding of the SDCC compiler and would have to change all these batch and version files in order to get the code to compile. This was outside the scope of this project and there was not enough time to do this.

4.3. Compiling

The Keil7 compiler should be installed to the default directory otherwise the batch files to compile the code must all be changed. The code for the phone should be placed in the directory `c:\PalmH323`. A number of batch files can be seen in this directory.

The first step in the compile process is to modify the `version.h` file in the `\inc` directory. Here one specifies which hardware type the phone has, what protocol is being used, as well as what language must be used on the display. The file contains a number of commented out `#define` statements for each different hardware type, protocol and language. To compile for a specific hardware type, the `#define` for that hardware type should be un-commented. For example, to compile for this phone (the PA168S), the line `#define PA168S` should be un-commented. The same applies for the protocol, which should be `#define SIP` for the SIP protocol, and for the language.

To clean all files from a previous compile `cleanall.bat` can be used. This will permanently delete all files in all directories that were created from a previous compile including all object files and hex files. It will not delete any source code or libraries.

The `setenv.bat` file should then be called. This file sets up all environmental variables and path information. If Keil7 is installed to a different directory to the default or if the phones code is not in `c:\PalmH323` then the correct locations should be put in this file.

To compile the libraries for which the source code is available, `makelibs.bat` can be used. Given no arguments it will attempt to compile all the libraries. To compile a specific library give the name of the library as the argument. For example, to compile the `P_UTIL` library enter the command `makelibs p_util`.

To compile each bank of code there is a file called `make.bat` in each of the directories. There are also various other files for generating hex files and linking everything together into a binary file. To simplify the whole process and to try and speed it up a batch file has been created that does everything requiring only one argument.

To use this batch file simply enter the command `makebin <protocol>` where `<protocol>` is the name of the protocol. To compile for SIP type `makebin sip`. This file automatically cleans all the directories and set up the environment variables. It will compile for the PA168s in English. The final binary file will be created in the root directory.

4.4. Updating firmware

Palmtool which is provided by the manufactures of the PA1688 chip is used to transfer the compiled binary file to the phone over the network. The phones IP address must be entered into Palmtool and the binary file to be transferred to the phone must be selected. When the update process begins, the phone enters an upgrade mode where it is unable to make or receive calls, the firmware is first transferred into the phones SDRAM. Once all the data has been transferred over the network the phone copies the firmware in the SDRAM to the flash memory and reboots.

The firmware can also be upgraded from the web interface of the phone. Once logged in there is an upgrade firmware button. The upgrade process occurs in the same way as above, however it takes about 3 times longer. The times for updating the firmware are listed below.

	Palmtree	Web Interface
Copying over network	59 sec	201 sec
SDRAM to Flash	30 sec	30 sec
Total	89 sec	231 sec

Table 1. Update Times

It can be seen in Table 1 that updating the firmware from the web takes more than two times longer than from Palmtree. Once the firmware has been transferred into the SDRAM, it takes the same amount of time to copy the data into flash memory. The difference in time between the two update methods must therefore be related to the way the data is transferred over the network.

4.5. Upgrade Issues

Whether updating from Palmtree or updating from the web interface there are some problems. After the firmware has been transferred to the SDRAM, the phone checks that the file that has been uploaded is actually a firmware file, and that it is the correct version for the phone. This however, does not work correctly, and even if the correct firmware file is loaded to the phone an error occurs. After transferring the firmware file over the network, Palmtree will display an error saying “invalid update method”. The phone must then be switched off and back on again. This will cause the data in the SDRAM to be lost and the phone will still have the old firmware.

One way to fix this problem is to set the debug mode of the phone to “no check” in the phone settings [11]. This causes the phone not to check the firmware file and overwrite the firmware in the flash memory. This can be dangerous as any firmware can be written to the phone. If the wrong firmware is loaded onto the phone and the

phone is unable to boot or go into upgrade mode then it will be impossible to correct the problem.

4.6. Failure recovery

When there are errors in the firmware the phone may not be able to boot to a stage where it is able to receive a new firmware file. In this case, the phone needs to be restarted in recovery mode, so that a working firmware file can be loaded back on to it. The remainder of this section describes what can cause the phone not to boot, and how the phone can be recovered.

There are sometimes errors in the code that the compiler will not pick up. These errors do however cause the phone to crash while it is booting up. During this project, this happened when variables were not cast to the correct type, particularly when strings were not cast as pointers to character arrays. The compiler did not give any warnings or errors, but when the compiled firmware was loaded onto the phone, it did not boot. When the phone has crashed during boot up, it will not be able to respond to any update requests and it will not be possible to access the phone from Palmtool or from the web interface.

To overcome this problem, there is a safety mechanism built into `page0`, which is the first bank of code that is loaded when the phone is switched on. The code in `page0` will check if the * key on the phone is pressed, and if so, the phone will go into recovery mode. When the phone is in recovery mode it will be able to respond to Palmtool. The errors in the code can then be fixed and the new firmware file can be transferred to the phone.

Sometimes the phone will not be able to get an IP address and therefore can not connect to the network. If this happens the phone can be switched off and switched on again holding the * key. This will reset the phones IP address to the default of 192.168.1.100. To access the phone, the host PC that Palmtool is on will also need to be configured to use the 192.168.1.* network.

During the upgrade process, the phone copies its settings from flash memory to SDRAM. The phone then copies the new firmware file into flash memory and finally, copies the settings back to flash memory. If there is an error in the firmware however, the phone sometimes can not copy the settings back to flash memory. If this happens, some of the phones settings will become corrupted, and the last 3 digits of the phone's mac address will all be set to zero's. In this case, the phone's settings should be set again and the mac address should be re-entered.

Chapter 5

Changes Implemented

This chapter describes the modifications made to the phone. The implementation for each modification will be explained. Finally some of the implementation challenges will be discussed.

5.1. Modifying the Display and showing the time

The first change to be made to the phone was the display. The display is used to give the user of the phone information about what it is doing and gives feedback when the user does something. If the user picks up the handset, the display should respond, displaying something such as 'make a call'. An understanding of how to update the display and where to update it was therefore essential for all the changes to be implemented.

5.1.1. Modifying message strings

A good start to improving the phone was to change some of the strings that were displayed on the screen. This was quite an easy task as all the strings are stored in a single file. The file is found in the following directory: `p_apps\lcd_en.c`. This is the file for all the English strings, if another language is to be used then the appropriate file for that language should be changed.

All the message strings are stored as character arrays of type `CCHAR`. The strings are thus located in the code memory space, or the flash memory. Any string declared in

this file should be less than 16 characters in length as this is the maximum length that will fit on the display. If a new string is added to the file it should also be declared in the header file located at, `inc\p_apps.h`.

From here it is easy to modify the strings that the phone displays on its screen. For example the string `g_cWaitLogon[]` can be changed from “Wait Logon” to something more user friendly such as “Connecting...”. Now wherever the phone used to display `Wait Logon`, it will now display `Connecting...`

5.1.2. UI update handler

After the phone has completed its initialisation and has connected to the server, the function `TaskLogonCompleted()` is called. Welcome messages can be put here as the function is only called once after the phone has booted. Instead of the phone showing its hardware version when it has finished booting a welcome message is now shown.

To do this one can remove the code to display the hardware version and call the `DisplayString` function as follows:

```
DisplayString((PCCHAR)"Welcome", 1);
```

This will display the string `Welcome` on the bottom line of the LCD display. The `DisplayString` function takes a pointer to a character array as its first parameter and therefore the string should be type cast to this type. The compiler will not pick up if the string is not type cast, however the phone will not boot when it is switched on. This will require the phone to be started in recovery mode and will need to be re-flashed. The welcome string above should rather have been declared in the `p_apps\lcd_en.c` file as a character array. This would allow it to easily be modified and would reduce the chance of any type casting errors.

When the phone is in its normal state it continuously runs through its main loop. Within this loop the function `UIHandleTimer()` is called on each repetition of the loop. This function checks for certain conditions such as whether there is a call in progress. The function also calls two other functions, `DisplayNormalState()` which

is called whenever a call ends or the handset is put down, and `CommonUIHandleTimer()` which is called when the phone is idle.

In the `CommonUIHandleTimer()` there is a timer called `Sys_lCurrentTime`. This timer can be used to make the phone show alternate messages on its display. The following code would alternate between showing “iLanga” and showing the phone number.

```
if (!(Sys_lCurrentTime % 20))
{
    DisplayLine(Sys_pLocalNumber, 1);
}
else if (!(Sys_lCurrentTime % 15))
{
    DisplayString((PCCHAR)g_cReady, 0);
}
```

5.1.3. Displaying the time

One goal of improving the display was to be able to show the time. When programming on a computer it is relatively easy to get the time, by simply calling a function that gets the time from the computers real time clock. In the Palm1 however, there is no real time clock, so the time cannot be obtained in this manner. It is also infeasible to try and use the controller’s clock as this runs in the tens to hundreds of megahertz. Even if keeping track of real time by using this clock was possible, the time would have to be reset each time the phone is switched on.

Since this is an IP phone, it will always be connected to the network, therefore a more efficient and accurate solution is to obtain the time from a time server. Rhodes University has a time server which supports the SNTP or simple network time protocol. When studying the code for the phone it was found that SNTP was already implemented.

The function that is used to send a request to the time server is `SntpDisplayHandleTimer()`. When this function is called, it stores the time value in a variable called `Sys_lCurrentTime`. To display the time and date on the display there is a function called `GetDateTimeDisplay`. This function has been modified to improve how the time value is formatted on the display. It takes two arguments, and is defined as follows:

```
PXCHAR GetDateTimeDisplay(BOOLEAN bLcdDisplay, UCHAR timeOnly)
```

The first argument is to say whether the time string is to be returned as a string buffer or to be written to a buffer in SDRAM. The second argument is to specify whether the time, date or both should be written to the display buffer. The function returns a type of `PXCHAR` which is a pointer to the temporary display buffer created in the function.

The SNTP handler also has a variable called `Sntp_bFinished` which will be set to true when the phone has received the time from the time server. Using this variable, one can display a different string if the phone has been unable to connect to the time server. The code below shows how the time can be shown on the display.

```
1     if (Sntp_bFinished)
2     {
3         DisplayLine(GetDateTimeDisplay(TRUE, 1), 1);
4         DisplayLine(GetDateTimeDisplay(TRUE, 2), 0);
5     }else{
6         DisplayLine(Sys_pLocalNumber, 1);
7     }
```

The `GetDateTimeDisplay` function returns a pointer to a character array, which is what the `DisplayLine` function takes as its first argument, and therefore no conversion or type casting is necessary. Line 3 will display the time on the bottom line of the display, line 4 will display the date on the top line of the display, and line 6 will display the phones extension number, should the phone fail to connect to the time server.

5.2. Missed calls

One feature that the traditional phone lacks is the ability to tell if there had been a call that was not answered. In cell phones this notification is commonly known as a missed call. The Palm1 does record all the calls that are missed and has a dedicated button for displaying these missed calls. The problem is that the phone does not record which of the missed calls are new, neither does it inform the user that there was a missed call. The user shouldn't be expected to have to check each time they come back to the phone to see if they missed a call. The phone should rather display if there are any new missed calls.

5.2.1. Storing miss call data

The phones missed call data is stored in its SDRAM in the area where dialled calls, and answered calls are stored. The space reserved for miss call data is 3840 bytes, which is for a maximum of 40 missed calls.

The first way that was thought of to display if there were any missed calls was to read this miss call data from SDRAM and then count how many missed calls there were. This would not work however, as the phone does not keep any record of which of these missed calls were new, and therefore the phone would always show that there were missed calls. One way to get around this would be to clear all the missed calls when the user presses the missed call button. This would not be ideal though, as sometimes details of old missed calls may be needed.

Another way to keep track of whether there are any new missed calls is with the use of a separate variable, which would be incremented each time the phone received a missed call and cleared when the missed call button was pressed. The value of this variable would then be displayed on the screen if it was greater than zero. This technique was implemented in the code by creating a new variable named `missed`. The variable was initialised to zero and was used as explained above. The phone should have shown that there was a missed call, but when tested, it did not. The problem was that the variable was created in the main segment of code. The variable was also incremented in another bank of code which handles receiving calls. During

the bank switching process it was possible that the variable was overwritten back to zero. One way of ensuring that a value will not be changed during the operation of the phone is to store it in SDRAM.

Using the same method as described above, a flag could be set in SDRAM to keep track of whether there had been a missed call. There is a block in SDRAM which is used to store flags, a reserved part of this block was used to store the missed call flag. Before the flag can be used, it should be initialised. The flag should be initialised to have a value of zero when the phone is starting up. This way of storing the flag did work and will be described in more detail below.

5.2.2. Showing if there is a missed call

As described in chapter 3 the SDRAM needs to be accessed through the DMA buffer. The function to read from SDRAM takes several arguments which are derived from the memory address that is to be accessed. The reserved memory space in SDRAM that is to be used has the address 0x000126. The high index is therefore 0, the low index is 1 and the length is 10. When reading the flag the DMA buffer should be offset by 26 hex, which is 38 in decimal. The following constants have thus been defined in case the missed call flag should change to a different memory address.

```
#define MISSCALLFLAG_HIGH      0
#define MISSCALLFLAG_LOW      1
#define MISSCALLFLAG_LENGTH   10
#define MISSCALLFLAG_CACHE    38
```

The following functions initialise the missed call flag to zero before it is ever read.

```
XBYTE[Sdram_pDataCache+MISSCALLFLAG_CACHE] = 0;
SdramWrite(MISSCALLFLAG_HIGH, MISSCALLFLAG_LOW, MISSCALLFLAG_LENGTH);
```

As mentioned above, the phone does record call data such as received and missed calls. All these functions are stored in main\callrec.c and this is a good place to modify the miss call flag. Within this file there is a function which is called whenever

the phone exits its call mode. If the user did not answer the call the call type will be set to be a missed call. The code below shows how the missed call flag is incremented if the last call was missed.

```
if (iType == CALL_TYPE_MISSED)
{
    //set the missed call flag in sdram
    SdramRead(MISSCALLFLAG_HIGH,MISSCALLFLAG_LOW,MISSCALLFLAG_LENGTH);

    if((USHORT)XBYTE[Sdram_pDataCache+MISSCALLFLAG_CACHE] < 15)
        XBYTE[Sdram_pDataCache+MISSCALLFLAG_CACHE]++;

    SdramWrite(MISSCALLFLAG_HIGH, MISSCALLFLAG_LOW, MISSCALLFLAG_LENGTH);
}
```

In the same file there is a function which is called when one of the call record buttons is pressed. Here it can be checked if the missed call button was pressed and if so, the missed call flag can be cleared to zero. Now that the flag is keeping track of how many missed calls there are, the number of missed calls needs to be displayed on the phone. There is a functions which displays a value with a prefixed string, but it would look better if the value was displayed before the string in this case. A function which does this was created, called `DisplayVal2`. A function called `isMissedCall` was also created to help determine if the missed call flag is above zero. The code below was written in the display update method and will show if there are any missed calls until the user presses the miss calls button, whereby the miss call flag will be set to zero.

```
if(isMissedCall()){

    SdramRead(MISSCALLFLAG_HIGH, MISSCALLFLAG_LOW, MISSCALLFLAG_LENGTH);
    if((USHORT)XBYTE[Sdram_pDataCache+MISSCALLFLAG_CACHE] == 1)
        DisplayVal2((USHORT)1, (PCCHAR)"Missed Call", 0);
    else
        DisplayVal2((USHORT)XBYTE[Sdram_pDataCache+MISSCALLFLAG_CACHE],
            (PCCHAR)"Missed Calls", 0);
}
```


5.3. Mute Button

Some phones have a mute button, which allows the user to talk without being heard on the other end of the line. This could be a useful feature when dialling into a conference room, which would allow the user to have a private discussion on their end, without interfering in the conference room.

In order to mute the input of the phone, a button would have to be pressed. There is a button on the phone with the description “Flash”, which is used to save speed dials. It was decided to use this button as the mute button as speed dial numbers can only be saved when the phone is idle, and the input volume should only be allowed to be muted while in a call. Because only one button was available for use, it would have to be a toggle button, muting and un-muting the phone.

5.3.1. Key Event Handler

The code for handling key presses can be found in the file `p_apps\key.c`, here each key is given a character value. The number keys are given their respective number characters. The function keys of the phone are given character values from ‘A’ upwards. If the key pressed was a function key, `HandleFunctionKeys()` will be called.

To find out which character is associated with which key the phone can be started in recovery mode. In this mode the phone will display the character associated with a certain key when it is pressed. In this way it was found that the “Flash” button had the character “J” associated with it.

The `HandleFunctionKeys()` function is found in `main\keypad.c`. The file actually has multiple declarations of this function, one for each hardware version of the phone. The one used when compiling will be the one whose hardware version is defined. Therefore the `HandleFunctionKeys()` which should be modified for the PA168s is the one which is located under the statement `#ifdefined PA168s`. Here, the

function to mute the input volume can be called if the key associated with the character “J” is pressed.

5.3.2. Muting the input Volume

Because the mute button is going to be a toggle button, the mute status will have to be kept in order to keep track of whether the input volume has been muted or not. A mute flag was thus created and stored in SDRAM, in the same way as the missed call flag was.

There are functions in the file main\ac97_0.c which are used to increase or decrease the output volume of the phone, however there was no function to set the input volume. When looking at the functions to change the output volume, it can be seen that a function `Ac97WriteReg` is being called. This function writes a value to a given register on the AC97 chip. To mute the input volume it can be called as follows.

```
Ac97WriteReg(AC97_RECORD_GAIN, AC97_MUTE_HI, AC97_MUTE_LO);
```

When the mute button is pressed the volume should be set to mute or back to its original volume, by writing to the AC97 chip. The display should be updated telling the user if the volume is being muted or un-muted, and the mute status flag should also be toggled. This should only happen when the phone is in call mode. The code required to do this is shown below.

```
if(IsCallMode())
{
    SdramRead(MUTEFLAG_HIGH, MUTEFLAG_LOW, MUTEFLAG_LENGTH);

    if((USHORT)XBYTE[Sdram_pDataCache+MUTEFLAG_CACHE] != 0){
        //set mute to off
        XBYTE[Sdram_pDataCache+MUTEFLAG_CACHE] = 0;
        SdramWrite(MUTEFLAG_HIGH, MUTEFLAG_LOW, MUTEFLAG_LENGTH);
        Ac97WriteReg(AC97_RECORD_GAIN, AC97_FULLVOL_HI, AC97_FULLVOL_LO);
        DisplayString((PCCHAR)"Mute Off", 1);
    }else{
        //mute the input volume
```

```

        XBYTE[Sdram_pDataCache+MUTEFLAG_CACHE] = 1;
        SdramWrite(MUTEFLAG_HIGH, MUTEFLAG_LOW, MUTEFLAG_LENGTH);
        Ac97WriteReg(AC97_RECORD_GAIN, AC97_MUTE_HI, AC97_MUTE_LO);
        DisplayString((PCCHAR)"Mute On", 1);
    }
}

```

The mute status is constantly displayed on the phone while the call is still in progress. This is done in the `UIHandleTimer` function by checking if the mute flag is set. If it is, the string “Volume Muted” is displayed on the screen. The user may however forget that the volume is muted when the call has ended and therefore the volume should be set to normal when a new call is started. In the file `main\vtask.c` there is a function `Call_Initialize()` which is called whenever a new call is started. Code required to set the input volume to normal was written in this function ensuring the volume is never left muted from a previous call.

5.4. Hiding the iLanga Password

The previous modifications have all been for general purpose uses of the phone, while this one is specifically for iLanga. It was discussed in the first chapter that iLanga can be set to ask the user for authentication when dialling out to the phone network. A way to conceal the password from being typed onto the screen was thus needed.

In order to conceal the password the phone would have to know which characters are part of the password. The way one enters their details when dialling externally is by entering their phone number, followed by their password, followed by the # key. Therefore a good way of identifying the password is to check if the last digits entered match the phone number. If so, the next characters entered should be replaced with a “*” and this should carry on until “#” is pressed.

Whenever a number key is pressed on the phone it is saved to a key buffer called `key_pBuf`. This buffer will contain the string of numbers that have been pressed since the call was started. The last key that was pressed can also be obtained from the

variable `Key_iCurrentID`. The function that is called when a number key is pressed is `handleNumberKeys()`, this function is responsible for tasks such as playing the DTMF tone associated with the number key. Changing the value of `Key_iCurrentID` to a “*” in this function would make sure that the password characters were not displayed, but would interfere with the operation of the phone. A better place to change the value of `Key_iCurrentID` would be just before it is saved to the key buffer.

There were two functions which were useful in comparing and finding the length of strings. They are `p_strlen_x` which returns the length of a string and `p_strcmp_x2x` which compares if two strings are equal. The problem with using the string compare function is that it can only compare two exact strings. A user may have entered other numbers before they were required to enter the phone number to dial externally. Suppose the phone number is 2000, if the key buffer contains the numbers 12342000, then the next character should be replaced with a “*”. However there is no function that returns a substring, and the string compare function can only compare exact strings. The string compare function does however take pointers to a character arrays as its arguments. The pointer to the key buffer could be moved to point at the correct place in the character array. The arithmetic to make the pointer point to the beginning of the phone number is $[\text{start of key buffer}] + [\text{length of key buffer}] - [\text{length of phone number}]$. The code below shows how this is done.

```
if(p_strlen_x(Key_pBuf) >= p_strlen_x(Sys_pLocalNumber))
{
    if(!p_strcmp_x2x((PXCHAR)(Key_pBuf+(p_strlen_x(Key_pBuf)-
        (p_strlen_x(Sys_pLocalNumber)*1))), (PXCHAR)Sys_pLocalNumber))
    {
        PassChars = 1;
    }
    if(PassChars)
    {
        if(Key_iCurrentID == '#')
            PassChars = 0;
    }
}
```

It can be seen that the number is not changed to a “*” immediately, but a flag is set, indicating that the number is actually a password character. In the function which saves the last pressed key to the key buffer, the flag is checked to see if the number should be replaced with a “*”. This can be seen in the code below.

```
iLen = p_strlen_x(Key_pBuf);
if(PassChars)
    Key_pBuf[iLen] = '*';
else
    Key_pBuf[iLen] = Key_iCurrentID;
```

When the “#” is pressed the password character flag will be cleared and the remaining numbers will be displayed as normal.

5.5. Features not implemented

One of the features of the phone to be improved was to use the status LED for something useful, rather than having it flash whenever the phone receives network traffic. The LED could have been used to indicate that there was a missed call instead of showing it on the display, or it could have been used to indicate that the phone’s input volume is muted in case the user forgets.

In the file `palm11lib\Led.c` there are functions to turn the LED on and off, as well as to make it blink. These functions are called throughout the boot process to indicate the status of the phone. When the phone is running through its main method these functions are not called at all, so another function must be calling the `Led_Blink` function. There is a call to the `TcpipHandleTimer()` function in the main loop which was believed to cause the LED to blink. This was an API call and the source code for the TCIP/IP library was not available. To try and stop the LED flashing a call to `Led_Off` was placed in the code directly after the call to the TCP/IP library. When the code was compiled and the firmware updated the LED continued to flash with no noticeable difference. The code inside the `Led_Blink` function to control the LED was then commented out and still the LED flashed in the same way.

The support staff of the PA1688 [12] were then asked about the problem and it was discovered that the LED is connected directly to the Ethernet chip. Because of this, there is no way to control the status LED from software. The LED can be stopped from flashing by disconnecting it from the Ethernet chip, but will not be able to serve any useful purpose.

Another feature that was planned was to add SIP voicemail message support. As mentioned in the first Chapter, Asterisk sends out a SIP notify message to a user's phone when they have received a voicemail. When investigating the SIP protocol implementation in a new version of the code for the phone, it was found that support for the Asterisk message waiting indicator (MWI) was already implemented. Code was added to make the phone display the string "New message" whenever an MWI was received from Asterisk.

This was tested with iLanga and a voicemail message was created. The user was sent an e-mail notification that they had received a message, however, the phone did not display that there was a new message. It was not known at this stage whether Asterisk was not sending the SIP packets, or whether the implementation in the phone was not correct. A packet capturer was then used with a soft-phone to see if the SIP packets were being sent. Packets were captured from when the phone call was started until after the user was notified by e-mail. No SIP notify packets were captured, this led to the belief that Asterisk was not sending the packets at all. When testing with a standalone version of Asterisk, that did not have SIP express router (SER) installed, the packets were sent, and the phone displayed that there was a new message.

From this it can be concluded that SER is not set up to forward the packets that are sent by Asterisk. Some configuration of SER will have to be done in order to allow the phone to receive these MWI's from iLanga.

5.6. Implementation challenges

It was mentioned in Section 4.4 that the time to update the phones firmware was quite a lengthy process. When including the time to set up the compiler and compile the

code and waiting for the phone to boot there is a large amount of time one has to wait from changing the code to seeing the results. If there is a problem with the phones firmware then the phone needs to be restarted in recovery mode and IP address settings need to be changed. This can make implementation quite slow.

To try and speed up the compile process a batch file was created that can compile the code with one command. The batch file is called `makebin.bat` and can be used as follows: `makebin %`, where `%` is the name of the VoIP protocol to be used, i.e. SIP, H323, IAX2, and MGCP. This takes about 20 seconds to compile the code and simplifies the compile process.

Chapter 6

Conclusion and Future Work

6.1. Conclusion

The documentation for the Palm1 was studied with regard to the hardware and software required to re-program it. A test version of the Keil7 compiler was installed and the code was successfully compiled. An overall understanding of the Palm1's code was achieved and this allowed the various modifications to the phone to be implemented. The implementation of these modifications also increased the understanding of some of the more detailed operations in the code such as memory access and event handling.

The Palm1 has also been modified to allow it to be integrated better as an end device to the iLanga system. The Asterisk MWI implementation was also successfully tested, however it was found that SER was not forwarding the SIP packets to the Palm1. The Palm1's negative features have been improved, making it a low cost, high value for money VoIP device.

6.2. Future Work

This section will explain any future work that can be done on the phone. This work will be partly based on the work done in this project.

6.2.1. Phone Security

A big security risk with the phone is that Palmtree can connect to any phone without authorisation; all that needs to be known is the phones IP address. This allows anyone with access to Palmtree to update any phones firmware. If an invalid firmware file is deliberately loaded, the phone could stop working. Palmtree also allows one to see and change any of the phones settings, including the phones account number and password. This would enable someone to steal these details and use another person's account, possibly using their credit to make phone calls.

There is a way to disable Palmtree from being able to access the phone at all. To do this, the phones debug mode setting must be set to disable. In this mode however the phone must be updated from the web, which is slow. It would also be a good feature for an administrator to be able to update phones on a network remotely using a common password. This would require the phone to check for a password before it accepts a connection with Palmtree. This would require modification of the code for the phone and for Palmtree for which the source code is available.

6.2.2. Encrypting the media stream

At the moment the phone uses the Real-time Transport Protocol (RTP) to transmit audio across the network. This data is not encrypted and could be captured off the network. A possible extension to this project would be to investigate and implement encryption for the phone, either using Secure RTP to encrypt the media stream, or IPSec to encrypt the IP packets being sent across the network. The phone would have to be tested to see if its processor can handle the encryption / decryption process.

References

- [1] Centrality Communications (2004). *PA1688 IP Phone Development guide*, Centrality Communications
- [2] Centrality Communications (2004). *PA1688 IP Phone Manual*, Centrality Communications
- [3] Cisco Systems (2005). *Cisco 7900 Series IP Phones*. Cisco Systems Inc 2005.
<http://www.cisco.com/en/US/products/hw/phones/ps379/>
- [4] Dutta, Sandeep. *Small Device C Compiler*. Sourceforge.net.
<http://sdcc.sourceforge.net/>
- [5] Dvorak, John C. Pirillo, Chris. Taylor, Wendy. (October 2003). *Online! The Book*. Prentice Hall PTR.
- [6] Keil Software. *LX51 User's Guide, Bank Switching*. Keil Software Inc 2005.
http://www.keil.com/support/man/docs/lx51/lx51_banking.htm
- [7] Linksys. *Feature-rich telephone service through your Internet connection*. Cisco Systems Inc 2005.
<http://www.linksys.com/products/product.asp?grid=33&scid=38&prid=651>
- [8] Mahy, R. (August 2004). *RFC3842, A Message Summary and Message Waiting Indication Event Package for the Session Initiation Protocol (SIP)*. Cisco Systems, Inc.
- [9] Penton, J., Terzoli, A. (September 2004). *iLanga: A Next Generation VoIP-based, TDM-enabled PBX*, South African Telecommunications Networks and Appliances Conference, Spiers

- [10] Rongrong, Lin. (May 2004). *PA1688 key-note speech on 5111soft VOIP*. Shanghai Business Conference. Centrality Communications
- [11] Rongrong, Lin. (2005). *PA1688 tech support*. Yahoo Groups.
<http://groups.yahoo.com/group/pa1688/message/2520>
- [12] Rongrong, Lin. (2005). *PA1688 tech support*. Yahoo Groups.
<http://groups.yahoo.com/group/pa1688/message/3713>
- [13] Swiss Voice. *IP Telephone Datasheet*. Swiss Voice.
http://www.swissvoice.net/ww/htm_ww/01_index_followup/follow_up.html
- [14] Wyss, Balz. (March 2003). *White Paper: Microsoft® Windows® CE .NET 4.2 – An Integrated, Flexible Software Platform for Building VoIP-Enabled Client Devices*. Microsoft Embedded and Appliance Platforms Group.