Contents

1 Project Overview....4

2 Project Introduction....5

2.1 Project aim....5

Telemetry unit (TU)....5

Remote Device (RD)....5

2.2 Technologies used....5

Bluetooth....6

General Packet Radio Service (GPRS)....6

Microcontroller....6

Java....7

3 Hardware and Software....8

3.1 Hardware....8

Microcontroller....8

Bluetooth Module....8

GPRS Modem....8

Remote device....9

3.2 Software....9

PIC Programming....9

Remote Device Programming....9

4 Basic Connectivity....10

4.1 Bluetooth....10

Telemetry Unit (acting as a server)....10

Remote Device (acting as a client)....10

4.2 GPRS....13

GPRS connection problems....13

Relay Server....13

Telemetry Unit....14

Remote Device (Client)....15

4.3 Simple Messenger Application....16

5 Wireless Telemetry System Specification....17

5.1 Connectivity....17

5.2 Functionality....17

Telemetry Unit....17

Remote Device....18

5.3 Data Transfer....18

Sending data – Remote Device to Telemetry Unit....19

Sending data – Telemetry Unit to Remote Device....20

6 Programming the Telemetry Unit....21

6.1 Object style approach....21
6.2 Data structures....21
Buffer....21
Circular_buffer....22
6.3 RS232 Communication....23
Receiving data....23
Sending data....24
6.4 Bluetooth and GPRS....25
Sending and receiving data....25
Initialising the GPRS modem....26
Processing received data....26
Connecting to the server....26
Sending an SMS....26
6.5 Sampling....27
Sample timer....27
Sending Samples....28
6.6 Main Program....28

7 Developing the Java Application....30

7.1 Java Application Design structure....30
7.2 Connection Package....30
Creating a Bluetooth connection....31
Creating an Internet connection....32
Sending an SMS....32
7.3 DataTransfer Package....32
DataReceiver....32
DataSender....33
7.4 UserInterface Package....34
J2ME Polish....34
Overview....34
Visual Appearance....35
Keeping the user updated....35
Handling received data....36
Sending data....36
Visualiser....36
7.5 Data flow diagram....37

8 The Relay Server application....38

8.1 RelayServer package....38
SocketRelay....38
Server....38
ServerStatus....39
ServerVisual....39

9 Telemetry System Data Flow....40

10 Testing....41

10.1 Telemetry Unit testing....41
Data Structures....41
RS232 Interfaces....41
Bluetooth and GPRS....41
Sample and Display timers....41
10.2 Remote Device application testing....41
10.3 Overall system testing....41
11 Telemetry System User Guide....43
11.1 Getting Started....43
Requirements....43
Installation....43
Navigation....43
11.2 Using the application....43
Connection Menu (Connecting to the telemetry unit)....43
Visualiser....44
Telemetry Unit Options....44
Remote Device Settings....45
12 Project Conclusion....49
12.1 Skills development....49
12.2 Wireless Telemetry System development....49
13 Appendix....50
13.1 Project Extension (Extra Features)....50
13.2 Project Construction....50
14 References....51

1 Project Overview

This report details the work into a small, portable, battery powered remote sensing unit that can be communicated to using both internet (over GPRS) and Bluetooth enabled devices. The control is bidirectional allowing runtime parameters to be changed at the device (with other users accessing the system being notified of configuration updates being transferred).

Data recorded from sensors on the system can be displayed graphically on all devices connected to the system. An additional extension to the system is the use of SMS alerts when specified levels are exceeded (for example the temperature of an experiment).

A typical scenario for the use of the system would be analyse the temperature fluctuations of a chemical reaction over several hours where continual monitoring by individuals is not possible (due for example to the climate in which the reaction is taking place).

The system in such a case would be connected to temperature sensors and placed where the reaction is running. Alerts are generated and sent to the user's mobile phone if the reaction exceeds a specified temperature limit; furthermore the facility exists to check on the reactions progress at any time by using a mobile phone or logging on to any available computer.

2 Project Introduction

This chapter describes the aims of the project undertaken and gives a brief introduction to the technologies used.

2.1 Project aim

The aim of this project was to design and build a wireless telemetry system. The system consists of two main elements, a telemetry unit and a remote device. The telemetry unit is a small, portable, easy to install unit. Sensory equipment feeds a signal into this unit. A remote device can then connect to the telemetry unit to analyse and monitor the signal.

This project concentrated on the development of the telemetry unit and a software application for remote devices rather than data acquisition from sensory equipment.

Figure 1 shows a block diagram of the telemetry system. A solid arrow represents wired data transfer, and a dashed arrow represent wireless data transfer.

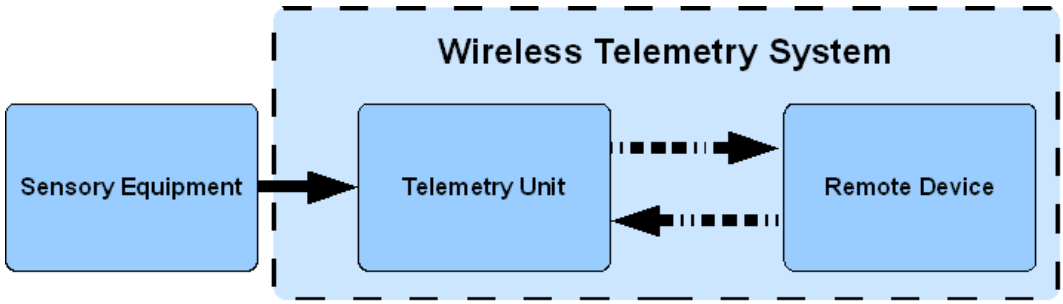


Figure 1: Wireless telemetry system block diagram

In this report telemetry unit may be referred to as the TU and remote device as RD.

Telemetry unit (TU)

The telemetry is a portable, battery powered unit. Sensory equipment is connected directly to the unit, however all data communication with remote devices is wireless. As a result only the telemetry unit itself is required to be in the monitored environment. The telemetry unit is small and easily portable so sensory equipment can to be placed wherever it is required, without having to worry about the requirements of large complex data processing devices such as desktop computers. .

Remote Device (RD)

Remote devices can communicate with the telemetry unit via a purposely designed software application. This application can run on many everyday electronic devices such as desktop computers, mobile phones and PDAs. Therefore communication is possible without a specifically designed device, resulting in reduced costs, greater flexibility and convenience for the end user.

2.2 Technologies used

Through research it was decided that a combination of Bluetooth, GPRS (General Packet Radio Service), microcontroller and Java technologies were to be used for developing the wireless telemetry system. This section introduces these technologies and gives reasoning as to why they were chosen.

Bluetooth

Bluetooth is a standard for short range, low power consumption wireless communication. It can achieve high data rates (up to 3 Mbps [1]) and uses relatively inexpensive hardware. Unlike infra red it does not required direct line of sight between transmitter and receiver. Many mobile devices such as laptops and mobile phones now come with Bluetooth connectivity as standard. The range of Bluetooth communication is between one and 100 meters depending on the class of device used. The majority of devices use class two Bluetooth limiting communication range to 10m.

A viable alternative to Bluetooth is Zigbee. Zigbee offers greater range at significantly lower power consumption compared to Bluetooth. However Zigbee has a lower data rate (maximum 250Kps [1]) and is still in its infancy. As a result very few devices cater for Zigbee, unlike Bluetooth which is very widespread.

General Packet Radio Service (GPRS)

Bluetooth has both low power consumption and high transfer rates, however its limitation is range. To increase the range of the telemetry system an alternative technology was required.

An ideal solution was to enable devices to communicate with the telemetry unit over the internet. Internet access is widely available and many common devices now have Internet connectivity. To give the telemetry unit internet access it could have been connected to an internet enabled network. Such networks are now commonplace in nearly all institutions, workplaces and homes. Wi-Fi chips can connect to these networks wirelessly, however they consume considerable power, require a wireless network to be present and complex configuration.

An alternative way to connect to the Internet is over GPRS. GPRS is a packet orientated mobile data service which connects to the Internet via mobile phone networks. It is much slower than both Wi-Fi and Bluetooth (data rates up to 171Kbps [2]) and data transfer is not free. However an internet connection can be made via GPRS wherever mobile signal is available, so no network infrastructure is required.

GPRS connectivity was incorporated in conjunction with Bluetooth connectivity. Bluetooth is used for short range communication as is is faster and data transfer is free. When long distance communication is required GPRS is available.

Microcontroller

The telemetry unit is controlled by a microcontroller. There are several key features which make microcontrollers ideal for this application

- Widely availability
- Low cost
- Low power consumption
- Re-programmability
- Ease of development

Micro controllers often have integrated features such as an ADC to sample the signal and a UART for serial communication. The microcontroller is connected to external Bluetooth and GPRS modules through which it communicates with remote devices.

Java

The software application run by remote devices was developed in Java. Java was chosen for its platform independence and the number of devices which currently support it. As it is platform independent a single application was developed which runs on any Java enabled device. Java is available for free on desktop computers and laptops, and is implemented on many modern mobile phones and PDAs

Other platform independent languages such as the Microsoft .NET platform are available, however Java is the most widespread.

3 Hardware and Software

Once these technologies had been chosen the associated hardware and software was required. This included a microcontroller, Bluetooth and GPRS modules and software packages.

3.1 Hardware

This section gives information about the hardware used during the development and construction of the wireless telemetry system.

Microcontroller

The microcontroller used was the Microchip PIC16F877 along with the PIC Millennium Development Board and PicStart Plus programmer. The PIC16F877 is a widely available 8-bit micro controller with an ADC for sampling the incoming signal and a Universal Asynchronous Receiver Transmitter (USART) for serial communication with the external Bluetooth module and GPRS modem. The development board provided a keypad, LCD, LEDs and a MAX232 chip to convert voltage levels for RS232 communication.

For more information on all these products please visit Microchips website [3].

Bluetooth Module

To enable the PIC to communicate via Bluetooth a Parani ESD200 Bluetooth module was used. This module was chosen because of its low power consumption, small form factor, simplicity of use and personal recommendation. The PIC communicates with this module through TTL level RS232 communication. The Parani ESD200 is configured through a PC wizard or by the PIC itself using AT commands. Once a Bluetooth connection has been established this module acts as a virtual serial port, sending and receiving raw bytes to and from the PIC and connected Bluetooth device.

For more information on the Parani ESD200 please visit [4].

GPRS Modem

The Telit TER-GX101S modem was used to enable GPRS internet connectivity for the PIC. It is a GPRS/GSM modem which can also make phone calls and send SMS messages. Like the Parani ESD the Telit TER-GX101S is configured through a PC wizard or by using AT commands and acts as a virtual serial port once a GPRS connection has been established.

The Telit TER-GX101S modem is not ideal for this purpose. It is a relatively large unit, requires an external aerial and a mains power supply. Therefore, although this modem was suitable for development, a GPRS module would be used for production of the telemetry unit rather than a GPRS modem. An ideal GPRS module would be one of the small, battery powered AarLogic C01/3 / C05/3 modules.

For more information on the AarLogic C01/3 and C05/3 GPRS modules please visit [5].

The Telit TER-GX101S modem is no longer in production, however for more information on its successor, the Telit TER-GX104 please visit [6].

Remote device

During development a single remote device was used. This was a Sony Ericsson K800i mobile phone. This is a standard mobile phone providing many common features including all of those required for this project – Bluetooth and GPRS Internet connectivity and Java ME support.

For more information on the Sony Ericsson K800i please visit the Sony Ericsson website [7].

3.2 Software

This section gives information on the programming languages and software packages used to develop the telemetry system.

PIC Programming

The PIC was programmed in C. The C programming language was chosen over assembly language due to its ease of development and programming clarity. The CCS C compiler was used along with Microchip MPLAB IDE. All source code was written using Notepad++, a free source code editor, for its features including syntax highlighting and tabbed document browsing.

For more information on and downloads for MPLAB, CCS and Notepad++ please visit [3], [8] and [9] respectively.

Remote Device Programming

A remote device can communicate with telemetry unit through a Java application. The remote device used for this project was a mobile phone (Sony K800i). Therefore the Java ME CLDC 1.1 MIDP 2.0 platform was used. This platform is available on most modern mobile phones and includes the wireless API required. Earlier Java ME platforms may also run the application, however this is not guaranteed. Java SE platforms (the Java platform run on desktop computers) require a separate GUI to be developed before the application will run. For more information on this please see chapter 7.

To help develop the application Netbeans IDE was used along with the Java Wireless Tool Kit (WTK). This enables applications to be developed, simulated and debugged on a PC without having to run the application on a mobile device. The Sony Ericsson SDK was also used on occasions.

To visually enhance the Java application J2ME Polish was used. This required the writing of Cascading Style Sheets (CSS) which are commonly used in website design. More information on J2ME Polish can be found in chapter 7.

For more information on Netbeans and J2ME Polish please visit [10] and [11].

4 Basic Connectivity

Once the hardware and software had been chosen, development began on the wireless telemetry system. This section covers the basics of creating a connection with, and sending data between, the telemetry unit and a remote device.

Once basic connectivity had been established a simple messenger program was developed which is documented at the end of this chapter (section 4.3).

4.1 Bluetooth

A Bluetooth connection consists of both a client and a server. In the case of this project the telemetry unit acts as the server and a remote device as a client.

Telemetry Unit (acting as a server)

The Parani-ESD Bluetooth module used for the telemetry unit can be in one of 4 modes:

- Mode0 – waiting for AT commands. Bluetooth is not active. Mode used for configuring the Parani-ESD.
- Mode1 – tries to connect to the last connected Bluetooth device.
- Mode2 – waiting for a connection form the last connected Bluetooth device.
- Mode3 – waiting for a connection from any Bluetooth device.

To act as a server the Parani-ESD is operated in mode3. In this mode it is discoverable by any device, unlike mode2 which only allows a single device to connect.

Once a client is connected to the Parani-ESD it informs the PIC by sending the message “CONNECT”. It then becomes a virtual serial connection, passing raw data to and from the PIC and connected device over the Bluetooth connection. This continues until the remote device disconnects, upon which the PIC is informed by a “DISCONNECT” message. The module automatically disconnects if the device goes out of range.

Messages to and from the PIC and Parani ESD are sent through the same RS232 interface as data from a remote device. Therefore these messages must be processed correctly, otherwise data or messages may be misinterpreted. The Parani ESD begins and ends each message with two command characters to allow messages to be distinguish from data.

Remote Device (acting as a client)

This sub-section describes how to make a Bluetooth connection in Java ME. All classes and methods described here are provided by the Java ME implementation.

The Java client side is considerably more complicated than telemetry unit server. Setting up the Bluetooth connection is achieved in three stages. These are:

- Device discovery
- Service search
- Service connection

A client device sends out a request to discover all available Bluetooth devices. Once a device has been discovered it can then be searched to determine the services it supports. There are a large range of Bluetooth services available but the service used by the Parani-ESD is the Bluetooth Serial Port Service. This service which allows the transfer of serial data between two devices.

Each service on any Bluetooth device has a universally unique identifier (UUID). This UUID can be used to construct a URL from which a connection can be created, in a similar fashion to creating a connection over the internet.Figure 2 Shows the URL of the serial port service of the Bluetooth dongle used during development.

```
btspp://00a09611fe32
```

Figure 2: Bluetooth URL

The protocol is btspp and 00a09611fe32 is the UUID. Once the connection has been made data can be sent freely between the two devices.

The Java package used to access the Bluetooth API is `javax.bluetooth`. Figure 3 illustrates the classes used to establish a Bluetooth connection.

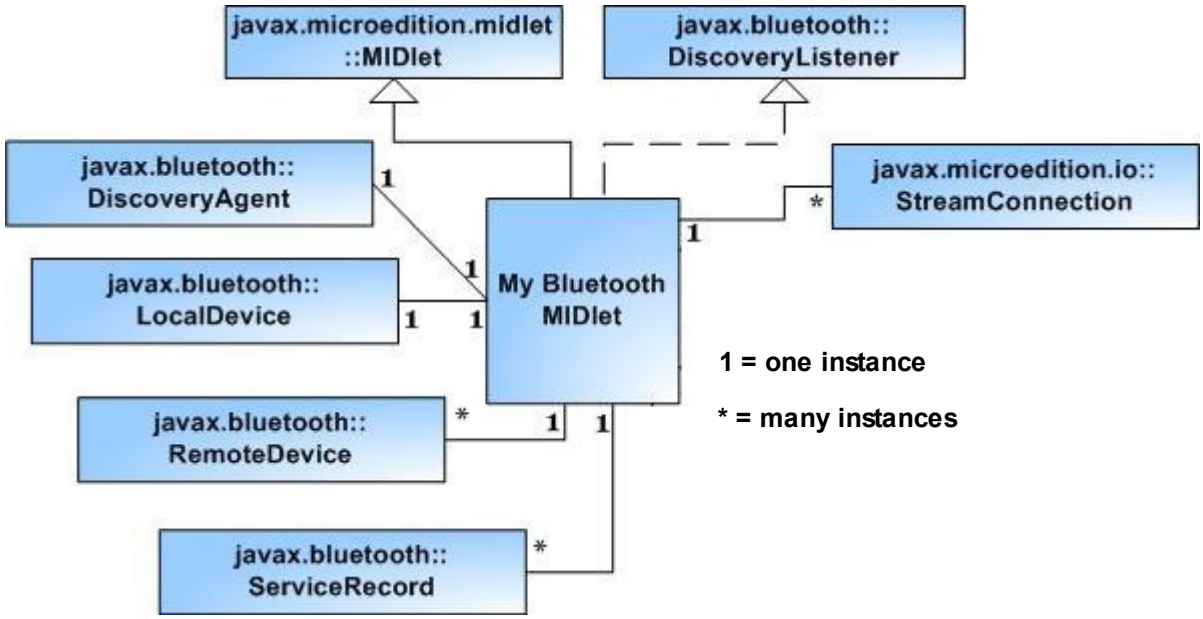


Figure 3: Classes used to establish a Bluetooth connection. [12]

LocalDevice

The `LocalDevice` class is a singleton (only ever one instance of the class) of which a reference is obtained by calling `LocalDevice.getLocalDevice()`. The `LocalDevice` instance provides methods to return the devices current Bluetooth settings (such as `getFriendlyName()`) and a method to return the `DiscoveryAgent` instance. It is the `DiscoveryAgent` which provides methods to search for devices and services.

DiscoveryAgent

The `DiscoveryAgent` is also a singleton, of which a reference is returned by calling the `LocalDevice.getDiscoveryAgent()` method. To begin device discovery (search for Bluetooth devices) a call must be made to the `DiscoveryAgent` method `startInquiry(..)`.

```
startInquiry(int accessCode, DiscoveryListener listener)
```

The `accessCode` argument should be `DiscoveryAgent.GIAC` which is a general inquiry access code which inquires about all devices. Other access codes can be used to discover only specific types of device. `Listener` is the instance of a class that implements the `DiscoveryListener` interface to which callbacks are made. Once a device has been discovered its services can be searched using the `DiscoveryAgent searchServices(..)` method.

```
searchServices(int[] attrSet, UUID[] uuidSet, RemoteDevice btDev,
DiscoveryListener listener)
```

The `uuidSet` argument is an array of UUIDs for the services of interest, in this case the UUID for the serial port profile. `BtDev` is the device to search and `listener` is the same as before. Both of these methods require considerable computation time, so rather than blocking they return instantly and communicate through call back routines defined in the `DiscoveryListener` interface implemented by `listener`.

DiscoveryListener

`DiscoveryListener` is an interface which defines the four call back routines shown in Figure 4.

```
deviceDiscovered(RemoteDevice btDevice, DeviceClass cod)

inquiryCompleted(int discType)

servicesDiscovered(int transID, ServiceRecord[] servRecord)

serviceSearchCompleted(int transID, int respCode)
```

Figure 4: DiscoveryListener interface

These routines must be implemented by any class which implements the `DiscoveryListener` interface. They are called each time a device/service is discovered and when the device/service search has completed.

RemoteDevice

For every device found a callback to `deviceDiscovered(..)` is made, providing a `RemoteDevice` instance representing the device found. This class provides methods to query the device for information such as its address, friendly name, authorisation and authentication. To discover which services the device offers a call must be made to the `DiscoveryAgent` method `searchServices()` with the `RemoteDevice` instance as one of the arguments.

ServiceRecord

Once `searchServices()` has been called the assigned listener will receiver a sequence of calls to `servicesDiscovered()` followed by a single call to `serviceSearchCompleted()`. Calls to `servicesDiscovered()` provide `ServiceRecord` instances for each service found. The `ServiceRecord` class provides the method `getConnectionURL()` which returns the URL of the service which can then be connected to using the `javax.microedition.io.Connector` class. The lines of code in Figure 5 connect to the URL specified in the string `url` and open both input and output streams which can be read from and written to accordingly.

```
StreamConnection conn = (StreamConnection) Connector.open(url);

InputStream is = conn.openInputStream();

OutputStream os = conn.openOutputStream();
```

Figure 5: Code to connect to a URL and open input and output streams

For more information on the Java Bluetooth APIs please visit [13].

4.2 GPRS

An Internet connection requires a server to listen for incoming connections and a client to connect to the server. Ideally the telemetry unit would act as a server waiting for a connection from a remote device. This required the telemetry unit to a open a server side socket on a designated port and any device could then connect to that socket. However several unforeseen problems prevented this.

GPRS connection problems

Whenever the telemetry unit enables GPRS it is assigned a different IP address. This meant that the remote device did not know what IP address to connect to. The same problem occurs when the mobile phone enables GPRS. This meant that neither the telemetry unit or remote device knew the others IP address, making communication impossible.

Originally it was thought that if the remote device or telemetry unit could open up a server side socket and then send its IP address in a SMS message then the other could connect to this address. Unfortunately there was another problem with this solution – each device could only retrieve its internal IP address, one that is not globally addressable (cannot be connected to).

It was evident that direct data transfer over GPRS was not going to be simple, if at all possible. The problem was equivalent to a hypothetical situation of two friends wanting to call each other, but neither knowing the others number, or even in fact their own.

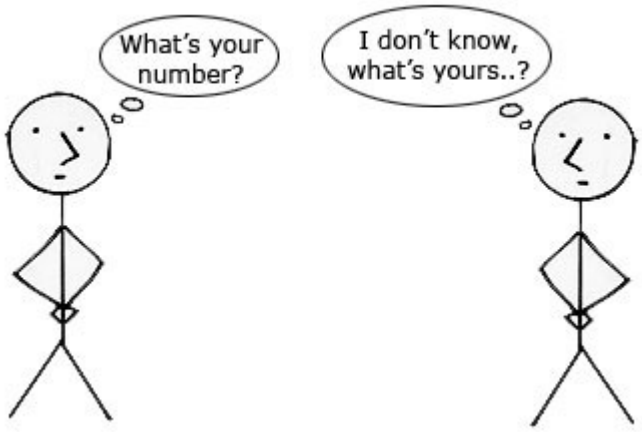


Figure 6: How do you ring a friend when you don't know their number?

They can send SMS messages to each other, however this is both very slow and costly, making it an unsuitable alternative.

The final solution was to introduce a third party – a relay server. Using the situation from before, this is a third friend whom both friends can ring and who will then pass on messages between them. This friend always has the same number, so it does not matter that the other friends do not know their own numbers.

Relay Server

The relay server opens a socket and waits until both the telemetry unit and a remote device have connected to it before directly relaying data between them. Neither the telemetry unit or remote device are aware of its presence. It can also be used to monitor Internet access, and keep records of any data transferred. The relay server must be hosted on the Internet. This can be done by purchasing a domain and leasing space on a web server, however this is costly. A free alternative is to use the services of DynDNS.

DynDNS offer free domains, and more importantly a free domain name service which maps a domain to a computer. This means that the server can be run on any computer which has internet access, removing the need to lease a dedicated web server.

Most internet users are assigned a dynamic IP address from their ISP. This IP address may change which would cause the domain name to no longer map to the required computer. However DynDNS provide a free program which runs on any computer (and quite a few routers) to ensure that the domain name is always mapped to the correct computer, regardless of its changing IP address. For more information on DynDNS please visit [14].

Telemetry Unit

The Telit GPRS/GSM modem accepts AT commands which are terminated by a carriage return character. The modem can be set to respond with either verbose responses such as 'OK' or numeric responses. It was found that numeric responses were easier to interpret than verbose responses. Figure 7 shows the numeric response codes and their interpretation.

Response Code	Interpretation
0	Command Successful
1	Connected
2	N/A (ring)
3	Disconnected
4	Command Error
5	N/A (no dial tone)
6	N/A (busy)
7	N/A (no answer)

Figure 7: Telit modem response codes

The Telit modem takes considerable time to processes even simple commands so a delay of 500ms was used in between all commands sent.

Making a GPRS connection

To connect to the relay server the Telit modem must connect to specified port on the server. Two commands are required to achieve this, `AT#SKTSET` and `AT#SKTOP`. The first sets the properties of the socket.

```
AT#SKTSET=0,21000,"WWW.DANNY.GOTDNS.COM"
```

This socket is set to be a TCP socket, which will connect to port 21000 on www.danny.gotdns.com. The `AT#SKTOP` command then opens the socket and starts the connection procedure.

```
AT#SKTOP
```

The default packet size, socket time-out and other socket properties can be set using commands which are specified in the Telit user manual available at [6].

Sending an SMS

To send an SMS using the Telit modem SMS messaging must be enabled using the command

```
AT+CNMI=2,1
```

SMS must then be set to 'Text Mode'.

```
AT+CMGF=1
```

The pseudo code to then send the SMS “Hello” to 07121212121 is shown in Figure 8.

```
Send : AT+CMGS= 07121212121

Wait to receive the '>' character

Send : Hello

Send : CRTL+Z (decimal 26) (end of message character)

Wait for message sent confirmation
```

Figure 8: Pseudo code to send an SMS using the Telit GPRS modem

Remote Device (Client)

The remote device used was a mobile phone which can also send SMS messages and connects to the relay server over GPRS.

Making a GPRS connection

Connecting to the relay server is very similar to connecting to a Bluetooth device, however the URL is previously known so no device or service discovery is required. Once a connection has been establish an `InputStream` and an `OutputStream` can be opened in exactly the same way as they are for a Bluetooth connection. The Java code in Figure 9 shows how to open a socket on port 21000 of www.danny.gotdns.com.

```
String url = "socket://www.danny.gotdns.com:21000";

StreamConnection conn = (StreamConnection) Connector.open(url);

InputStream is = conn.openInputStream();

OutputStream os = conn.openOutputStream();
```

Figure 9: How to open a socket in Java

Sending an SMS

To send an SMS a URL is required. This URL is “sms://” followed by the recipients phone number. Figure 10 shows how to send a SMS in Java ME.

```
String url = "sms://07121212121" ;

String message = "Hello";

// opens connection
```

```
MessageConnection conn = (MessageConnection) Connector.open(url);

// prepares text message
TextMessage sms = (TextMessage)
conn.newMessage(MessageConnection.TEXT_MESSAGE);

// write the message
sms.setPayloadText(message);

// send the sms
conn.send(sms);
```

Figure 10: How to send an SMS in Java ME

4.3 Simple Messenger Application

A simple messenger application was developed to test the transfer of data between the telemetry unit and a remote device. Once a connection is established it appears identical to both the telemetry unit and remote device whether it is a Bluetooth or GPRS connection. Therefore the messenger application can be run over either connection, however it was developed and tested for a Bluetooth connection to omit the added complexity of developing a relay server at this stage.

One-way messenger

The original messenger application allowed the remote device to search for Bluetooth devices and connect to them. Once connected to the telemetry unit messages could be written and sent over the Bluetooth connection. They were then displayed on the development board LCD.

Two-way messenger

The application was then extended by turning the development board keypad into a fully functioning keypad capable of typing standard alpha-numeric characters (similar to a mobile phone keypad). Messages were then typed on the keypad and sent from the telemetry unit to the remote device. This added extra complexity as both the telemetry unit and remote device were now sending and receiving messages simultaneously. This required the PIC to use interrupts to receive data and the remote device application to have a separate thread for receiving data.

Sample sending

Once the two way messenger was fully functional the application was extended again. This time the telemetry unit sent messages at the same time as sending regular readings from the PICs ADC. The ADC and a timer interrupt were used to send readings at a specified time interval. The remote device application was also to display both messages and the samples being sent. The messages were displayed in a main window and samples in the windows title bar. When receiving data, messages and samples were differentiated between by using a different header byte for each.

Overall Outcome

Developing this application proved the wireless telemetry system was achievable. Connection establishment, data transfer and signal sampling were all implemented successfully. Through achieving this key programming skills and routines were developed. These included multi threading, use of interrupts, automated data processing/buffering, routines to operate the development board keypad and LCD and simple Java ME graphical user interface construction.

5 Wireless Telemetry System Specification

Once basic connectivity and data transfer had been achieved a full specification for the wireless telemetry system was produced. This section documents the specification.

5.1 Connectivity

The telemetry unit supports both Bluetooth and Internet connectivity. This is limited to a single Bluetooth and a single Internet connection, however both connections are supported simultaneously. This allows two remote devices to communicate with the telemetry unit at the same time.

Bluetooth connections connect directly to the telemetry unit whereas Internet connections are via a relay server. The telemetry unit connects to the server via GPRS. Due to the costs involved with GPRS the telemetry unit is not constantly connected to the server. Therefore to initiate an internet connection an SMS message is sent to the telemetry unit requesting for it to connect to the relay server. This process is displayed graphically in Figure 11.

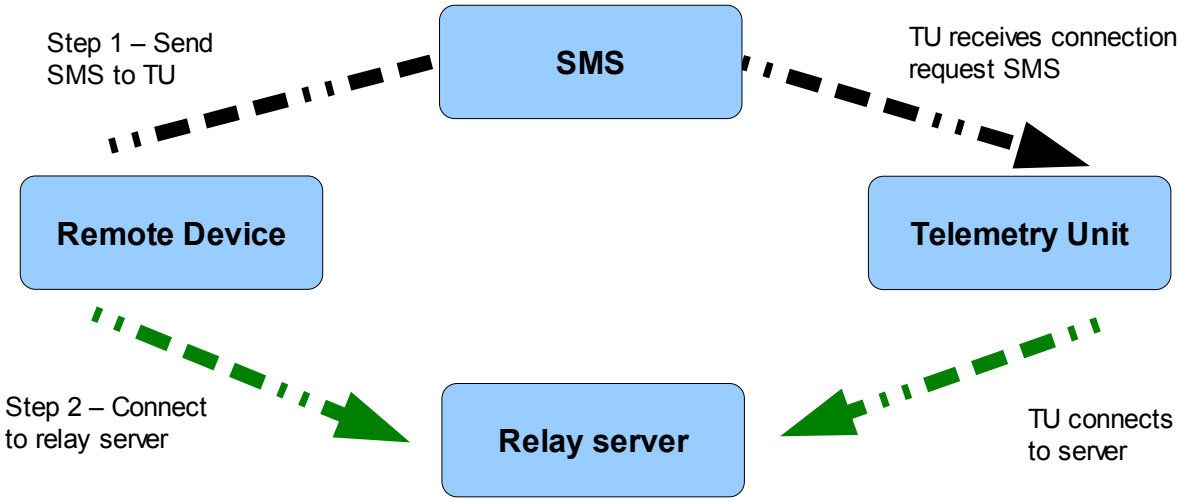


Figure 11: Process of connecting to the telemetry unit over the Internet

5.2 Functionality

This section describes the functionally of both the telemetry unit and the remote device application.

Telemetry Unit

The telemetry unit is able to:

- Sample an incoming signal between 0-5V with 8 bit resolution
- Support sample rates from micro seconds to hours
- Send live samples to all connected devices
- Send samples at a different rate to the sample rate
- Send a warning SMS to a remote device if this signal reaches a pre-defined level (critical level) while no device is connected

Remote Device

The remote device application is able to:

- Display the value of incoming samples
- Graphically display incoming samples
- Change the telemetry unit sample rate
- Change the rate at which the telemetry unit sends live samples (display rate)
- Change the critical level
- Warn the user if the critical level has been reached
- Change to phone number of the device to which the telemetry unit sends critical level warnings
- Change the server address to which it connects
- Change the server address to which the telemetry unit connects
- Change the phone number of the telemetry unit (to which an internet connection request SMS is sent)

5.3 Data Transfer

Data is be transferred in packets. These packets are the same regardless of the connection type. Each packet contains one of the following:

- Request for the current value of a setting (RD -> TU)
- Request to change a setting (RD -> TU)
- The current value of a setting (TU -> RD)
- Value of the latest sample (TU -> RD)

When a remote device connects to the telemetry unit it does not know the value of the telemetry unit's settings (for example sample rate). Therefore the remote device sends individual requests for the value of each setting. The telemetry unit then sends back the value of each setting. The remote device can change any setting by sending a request to change the setting along with the new value. Once this change has been successful the telemetry unit sends the new value of the setting back to the remote device. This is shown graphically in Figure 12.

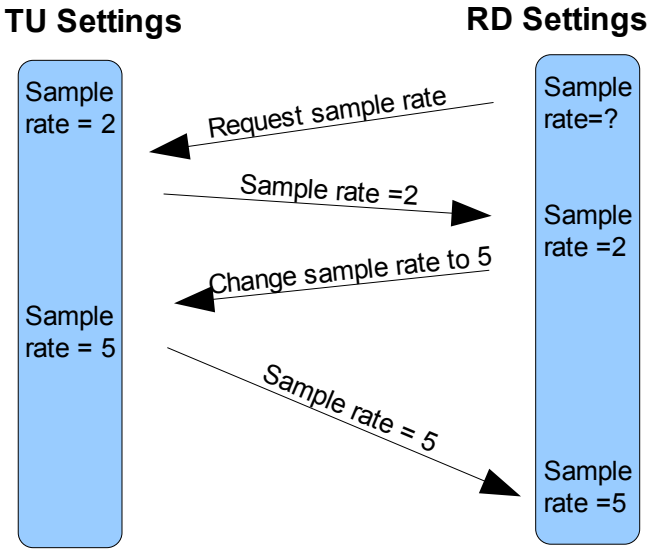


Figure 12: Requesting and changing the telemetry unit's settings

The remote device assumes that the setting has not been changed unless it receives the updated setting. This is to prevent a user from thinking that they have changed a setting when the change failed. If more than one device is connected to the telemetry unit then both devices are sent the updated setting automatically.

Sending data – Remote Device to Telemetry Unit

Data sent from a remote device to the telemetry unit is in a packet structure as shown in Figure 13.

BYTE 1	**BYTE 2**	**BYTE 3**	**BYTE 4**	**BYTE .....**
DATA_HEADER (0xFF)	CMD_HEADER	CMD_DATA		

Figure 13: Remote device to telemetry unit data packet structure

DATA_HEADER is a single byte of decimal value 255 (hex FF) which informs the telemetry unit that this is the start of a valid data packet. This header enables the telemetry unit to separate data sent from a connected device to messages sent by the Bluetooth or GPRS modules themselves (such as a connection or disconnection notification). It also prevents erroneous data from being interpreted as commands, as all commands must be preceded by the DATA_HEADER. The next byte is the CMD_HEADER. This tells the telemetry unit what action is required and what (if any) data follows. The command headers used are shown in Figure 14.

CMD_HEADER	**Value**	**Description**
HEADER_SAMPLERATE	0x01	Request for the current sample rate. No CMD_DATA supplied.
HEADER_DISPLAYRATE	0x02	Request for the current display rate. No CMD_DATA supplied.
HEADER_PHONENUMBER	0x03	Request for the phone number stored to which alert messages are sent. No CMD_DATA supplied.
HEADER_CRITICALLEVEL	0x04	Request for the current critical level value. No CMD_DATA supplied.
HEADER_SERVERURL	0x05	Request for the URL of the server to which the PIC connects when a GPRS connection request is received. No CMD_DATA supplied.
HEADER_SET_SAMPLERATE	0xF1	Request to change the sample rate. CMD_DATA = sample rate.
HEADER_SET_DISPLAYRATE	0xF2	Request to change the display rate. CMD_DATA = display rate.
HEADER_SET_PHONENUMBER	0xF3	Request to change the phone number stored. CMD_DATA = phone number.
HEADER_SET_CRITICALLEVEL	0xF4	Request to change the critical level. CMD_DATA = critical level

HEADER_SET_SERVERURL	0xF5	Request to change the server URL stored. CMD_DATA = server URL.

Figure 14: Data headers

Sending data – Telemetry Unit to Remote Device

Data sent from the telemetry unit to the remote device is wrapped in same packet structure as data sent from the remote device to the telemetry unit.

For SET commands an attempt is made to change the requested setting with the data provided. The resulting value of that setting, whether the change has been successful or not, is then sent back to the remote device. The command headers used are shown in Figure 15.

CMD_HEADER	**Value**	**Description**
HEADER_SAMPLE	0x0A	The following CMD_DATA is the latest sample
HEADER_SAMPLERATE	0x01	The following CMD_DATA is the current sample rate.
HEADER_DISPLAYRATE	0x02	The following CMD_DATA is the current display rate.
HEADER_PHONENUMBER	0x03	The following CMD_DATA is the phone number stored to which any alert message will be sent.
HEADER_CRITICALLEVEL	0x04	The following CMD_DATA is the current critical level value.
HEADER_SERVERURL	0x05	The following CMD_DATA is the URL of the server to which the PIC will connect to when a GPRS connection request has been received.

Figure 15: Data headers

6 Programming the Telemetry Unit

This section goes into detail about the software developed to run on the PIC microcontroller. This software implements the functionality described in the previous chapter.

6.1 Object style approach

The C programming language is a procedural language. However all code has been developed using an object orientated approach. This allows for easily readable and changeable code. It helped to reduce potential errors and allowed for each 'class' to be tested separately.

Classes were constructed using structures, and their associated methods are functions which take a pointer to the instance of the class structure as an argument. Figure 16 shows and example of the `method1(..)` method of the `myclass` class.

```
myclass_method1(struct myclass * instance, int agr1, int agr2);
```

Figure 16: Example of a class method

6.2 Data structures

For convenience, coding simplicity and to avoid memory leaks two simple data structures were developed. These are `buffer and circular_buffer`.

Buffer

`Buffer` is a simple `struct` containing a pointer to a standard C array and two integers which store the size of the buffer (number of bytes allocated in the data array) and the index of the next empty byte in the array.

```
struct buffer {

     BYTE * data;

     int size;

     int next_in;

} ;
```

Figure 17: Buffer structure

These variables should not be accessed directly. Instead a pointer a `buffer` instance should be passed to the specifically defined functions found in D_BUFFER.C. Functionality is provided to dynamically allocate and initialise a `buffer`, read and write bytes to a `buffer`, and query the current status and size of a `buffer`. The buffer API is shown in Figure 18.

```
struct buffer * buffer_init(struct buffer * buff, unsigned int size);

struct buffer* buffer_from_string(struct buffer * buff, BYTE *string);

BYTE buffer_read_byte(struct buffer * buff, unsigned int byte_num);

int1 buffer_write_byte(struct buffer *buff, BYTE data);
```

```
int buffer_write_bytes(struct buffer * buff, BYTE * src, int size);

int buffer_write_string(struct buffer * buff, BYTE * string);

void buffer_clean(struct buffer * buff);

void buffer_reset(struct buffer * buff) ;

int1 buffer_full(struct buffer * buff);

int buffer_space(struct buffer * buff);

void buffer_display_data(struct buffer * buff);

void buffer_display_idata(struct buffer * buff);
```

Figure 18: Buffer API

More detailed information and the implementation of each of these functions can be found in D_BUFFER.C.

Circular_buffer

`Circular_buffer` is an implementation of a circular buffer with FIFO access. The structure is shown in Figure 19.

```
struct circular_buffer {

     BYTE * data;

     int size;

     int current_size;

     BYTE next_in;

     BYTE next_out;

     int1 full;

     int1 empty;

} ;
```

Figure 19: Circular_buffer structure

As with the `buffer` structure these variables should not be accessed directly. Similar functions are provided for `circular_buffer` as for `buffer`. The API is shown in Figure 20.

```
struct circular_buffer * circular_buffer_init(struct circular_buffer *
buff, unsigned int size);

int1 circular_buffer_write_byte(struct circular_buffer
*mycircular_buffer,BYTE data);

BYTE circular_buffer_read_byte(struct circular_buffer *
mycircular_buffer);

int circular_buffer_read_string(struct circular_buffer *
mycircular_buffer, BYTE * string);
```

```
int circular_buffer_read_data(struct circular_buffer *
mycircular_buffer, BYTE * dest, int size);

int circular_buffer_write_data(struct circular_buffer *
mycircular_buffer, BYTE * src, int size);

int1 circular_buffer_full(struct circular_buffer * mycircular_buffer);

int1 circular_buffer_empty(struct circular_buffer *
mycircular_buffer);

int circular_buffer_space(struct circular_buffer * mycircular_buffer);

void circular_buffer_clean(struct circular_buffer *
my_circular_buffer);

void circular_buffer_display_data(struct circular_buffer *
mycircular_buffer);

void circular_buffer_display_idata(struct circular_buffer *
mycircular_buffer);
```

Figure 20: Circular_buffer API

More detailed information and the implementation of each of these functions can be found in D_CIRCULAR_BUFFER.C.

By using these data structures and associated methods data handling and storage responsibilities are separated from the main program. This simplifies the main program and eases code alteration. These structures were tested separately to ensure their validity. This eased later debugging.

Both `buffer` and `circular_buffer` have been implemented to ensure that memory leaks will not occur. An attempt to write too much data to either of these buffers results in the data being discarded and notification via function return value.

6.3 RS232 Communication

The PIC communicates with the Bluetooth module and GPRS modem via RS232. To enable simultaneous communication two RS232 communication interfaces were developed, `RS232_COMM1` and `RS232_COMM2`. These can be found in the corresponding D_RS232_COMM1.C and D_RS232_COMM2.C files. Both interfaces have identical API, however `RS232_COMM1` utilises the PICs USART (a hardware device designed to convert parallel data for serial transmission and vice versa) whereas `RS232_COMM2` handles serial data by using software interrupts. This is less reliable and more processor intensive than using the USART, however the PIC16F877 only has one USART. Unfortunately the only PIC models which have more than one USART have more than 40 pins and were not compatible with the development board or programmer used.

Receiving data

Both interfaces automatically receive incoming data via interrupt routines and store it in a `circular_buffer`, of size defined by `RS232_COMM_BUFFER_SIZE`. Buffered data can then be retrieved by using the corresponding `rs232_comm_get_byte()` function which returns the next byte and removes it from the buffer. `Rs232_comm_get_byte()` returns immediately and should not be called without first checking that new data is available (eg using the `rs232_comm_data_available()` function). If `rs232_comm_get_byte()` is called when there is

no data available the returned byte is invalid. The buffer should not be accessed directly and if the it becomes full then any data received will be discarded until space becomes available.

In many applications the PIC expects more data to arrive over RS232, for example if three digits of a phone number had arrived then eight more digits are expected. However the PIC processes data faster than it arrives over RS232. This results in the PIC having to wait for the remaining data. `Rs232_comm_wait_for_data()` is a function which will wait indefinitely and only returns once data is available. However use of this function is not advised as data may have been lost or corrupt and never arrive, resulting in the program stalling indefinitely.

To solve this problem `rs232_comm_wait_ms_for_data(unsigned long ms)` was developed. This function returns `TRUE` immediately if there is already data available in the buffer. If data is not available then it waits a maximum of `ms` milliseconds for data to arrive. If no data arrives by the end of this period then it returns `FALSE`. This prevents the program from waiting indefinitely for data. This function returns `TRUE` as soon as data is available and does not wait the time specified before returning.

To ensure that the receive buffers do not fill up `rs232_comm_data_available()` is polled regularly and the data processed as soon as possible.

Functions for both RS232 interfaces were developed to receive a specified number of bytes and return them in a `buffer` structure. A standard time-out is used to receive the bytes. If the time-out expires then the function returns the number of bytes received.

Sending data

Functions for both interfaces were developed to send a single byte or an entire `buffer` over RS232.

To help debug and visualise data being sent over RS232, both interfaces define pins which are set high/low (ideal for connecting to LEDs) when data is available and when the receive buffers are full. These pins can be changed by altering the `#define` statements at the top of the corresponding .C files. The full API for each communication interface is shown in Figure 21.

```
int1 rs232_comm1_init();

BYTE rs232_comm1_get_byte();

unsigned int rs232_comm1_get_bytes(struct buffer * buff, unsigned int
bytes);

int1 rs232_comm1_send_byte(BYTE data);

int1 rs232_comm1_send_buffer(struct buffer * buff);

int1 rs232_comm1_data_available();

void rs232_comm1_wait_for_data();

int1 rs232_comm1_wait_ms_for_data(unsigned long ms);

void rs232_comm1_clear_buffer();

void rs232_comm1_display_data();

void rs232_comm1_display_idata();
```

Figure 21: Rs232_comm1 interface API

More detailed information and the implementation of these functions can be found in D_RS232_COMM1.C / D_RS232_COMM2.C.

Figure 22 shows the flow of data in and out of the PIC via RS232 using these interfaces.

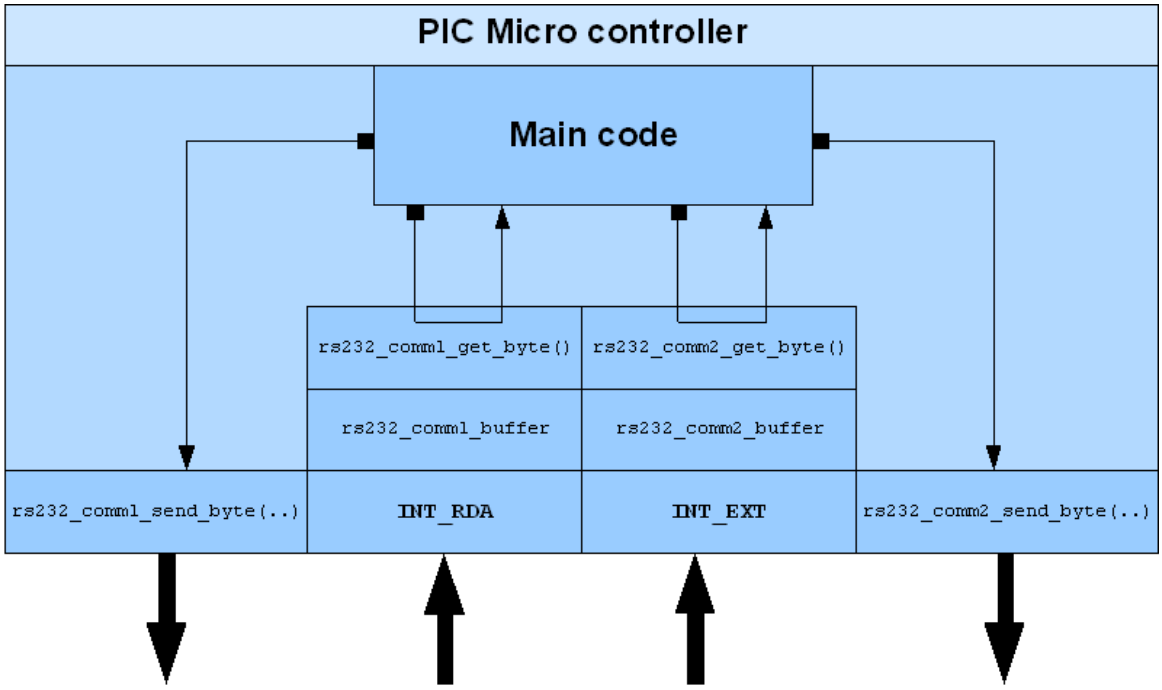


Figure 22: Diagram showing the flow of data in and out of the PIC using the Rs232 interfaces.

6.4 Bluetooth and GPRS

D_BLUETOOTH.C and D_GPRS.C were developed over the top of the `rs232_comm` interfaces. This section goes into more detail about D_GPRS.C however the same principles apply to the functions in D_BLUETOOTH.C

Sending and receiving data

The functions shown in Figure 23 were developed to send to and receive data from the GPRS module.

```
BYTE gprs_recv_byte()

int gprs_recv_bytes(struct buffer * buff, int num_bytes)

int1 gprs_send_byte(BYTE data)

int1 gprs_send_buffer(struct buffer * buff)

int1 gprs_wait_ms_for_data(unsigned long ms)
```

Figure 23: GPRS functions

These functions call the relevant rs232_comm2 functions. For example `gprs_send_byte(..)` calls `rs232_comm2_send_byte(..)`. This makes the code easier to read and allows the underlying functions could be changed easily. For example the GPRS module could be moved to `rs232_comm1` by simply changing these functions to the `rs232_comm1` equivalents.

Initialising the GPRS modem

To ensure that the GPRS modem is operational `gprs_init()` was developed. This function attempts to set the modem correctly and returns `TRUE` if this was successful. This is called before using any of the other `gprs` functions.

Processing received data

Once data has been received it is analysed to determine if it is data that has been sent to the telemetry unit from a remote device or if it is a message from the GPRS modem itself. To do this `gprs_process()` was developed. This function returns a `GPRS_RESPONSE enum` which indicates what has been received. The `GPRS_RESPONSE enum` is defined in D_GPRS.C and shown in Figure 24.

```
enum GPRS_RESPONSE {

     GPRS_RESPONSE_OK,
     GPRS_RESPONSE_ERROR,
     GPRS_CONNECT,
     GPRS_DISCONNECT,
     GPRS_GOT_SMS_PROMPT,
     GPRS_SMS_RECEIVED,
     GPRS_SMS_SENT_OK,
     GPRS_DATA_HEADER,
     GPRS_UNKNOWN_HEADER

};
```

Figure 24: GPRS_RESPONSE enum

By developing the `gprs_process()` function the GPRS modem can be used without having to worry about deciphering messages it may send and separating them from data sent by a remote device. This function simply returns `GPRS_DATA_HEADER` if the data is from a remote device or a different `GPRS_RESPONSE` depending on the message sent by the modem. This hides the operation of the modem from the main code. The modem can therefore be changed with minimal disruption, as only the implementation of this function will need altering.

Connecting to the server

The `gprs_connect_to_server()` function connects to the relay server. The function itself does not return a value and does not state whether the connection was successful or not. The modem often takes considerable time to return a connection established or failed message and so it was decided that this function should not wait this amount of time before returning. The status of the connection is determined by calling `gprs_process()` once data is available. If the modem has successfully connected to the server then `GPRS_CONNECT` is returned, and if not then `GPRS_DISCONNECT` or `GPRS_RESPONSE_ERROR` is returned.

Sending an SMS

To send an SMS `gprs_send_sms(struct buffer* number, unsigned int level)` was developed. This function takes two arguments, a `buffer` containing the phone number to send the SMS to and an integer which represents the critical level that the input signal reached. This function takes several seconds to return as sending an SMS takes considerable time, however it does not wait to determine whether the SMS was sent successfully. As with connecting to the server this is determined by calling `gprs_process()` once data is available and

GPRS_SMS_SENT_OK is returned if the SMS was sent, and GPRS_RESPONSE_ERROR if not.

6.5 Sampling

To sample the ADC two interrupts are used – INT_TIMER1 and INT_AD. INT_TIMER1 is set to trigger at the required sample rate, and calls read_adc(ADC_START_ONLY). This function starts the analogue to digital conversion process but returns immediately to prevent the PIC from spending unnecessary time in the INT_TIMER1 interrupt routine. Once the conversion is complete the INT_AD interrupt is triggered. This routine retrieves the digital value and stores it appropriately. To ensure INT_TIMER1 does not flood the ADC with requests this interrupt is disabled in it own routine, and only enabled again once the conversion is complete.

Sample timer

To change how often samples are taken the function setup_sample_timer(long multiply, SAMPLE_TIME time) was written. This accepts a multiplication factor and a SAMPLE_TIME enum. This is the time unit to be multiplied to achieve the required sampling interval. The possible time units developed are shown in Figure 25.

```
enum SAMPLE_TIME {

     SAMPLE_TIME_10us,
     SAMPLE_TIME_100us,
     SAMPLE_TIME_1ms,
     SAMPLE_TIME_10ms,
     SAMPLE_TIME_100ms,
     SAMPLE_TIME_1s

};
```

Figure 25: SAMPLE_TIME enum

The function sets the INT_TIMER1 interrupt according to the supplied arguments. Setting the timer is complex so timer values required for each SAMPLE_TIME are hard coded. INT_TIMER1 triggers at the SAMPLE_TIME set, however read_adc(ADC_START_ONLY) is only called once the timer has been triggered multiply times, achieving the required sample time. This means that although calling setup_sample_timer(10, SAMPLE_TIME_1ms) and setup_sample_timer(1, SAMPLE_TIME_10ms) result in samples being taken with the same time interval, setup_sample_timer(1,SAMPLE_TIME_10ms) is more efficient as INT_TIMER1 is only triggered once per sample rather than ten times.

Setting TIMER_1

TIMER_1 is a 16 bit counter which triggers INT_TIMER1 once it over flows. The counter can be set to increment every one, two, four or eight instruction cycles (one instruction cycle is equal to four oscillator periods). The value from which counting begins can also be set. By changing these values INT_TIMER1 is triggered at the required time interval. Calculations determining the correct TIMER_1 setup values for each SAMPLE_TIME are described in the following paragraph.

```
oscillator period = 1 /  8MHz = 0.125us

1 instruction cycle = 4 x oscillator period = 0.5us
```

SAMPLE_TIME_10us = 20 instruction cycles. Therefore let TIMER_1 count to 20, incrementing once every instruction cycle. To make TIMER_1 overflow after counting to 20 set the timer to

start counting from 65515, as it will overflow after 65535 (maximum for a 16 bit counter) has been reached.

SAMPLE_TIME_100us = 200 instruction cycles. This can be achieved by various methods. By setting the timer to increment once every 8 instruction cycles, TIMER_1 must count to 25 before triggering. Therefore the timer must begin counting at 65510.

SAMPLE_TIME_1ms = 10 x SAMPLE_TIME_100us. TIMER1 can be setup the same as before, however this time it must count to 250 rather than 25. (Begin counting at 65285).

SAMPLE_TIME_10ms = 10 x SAMPLE_TIME_1ms. Count to 2500.

SAMPLE_TIME_100ms = 10 x SAMPLE_TIME_10ms. Count to 25000.

SAMPLE_TIME_1s = 10 x SAMPLE_TIME_100ms. Count to 250000. However TIMER_1 can only count to 65535 as it is a 16bit counter. Therefore the timer is set to trigger every 0.25s, (count to 62500), however it waits until it is triggered four times.

Sending Samples

When a device is connected to the telemetry unit samples are sent for monitoring. There are many occasions when it would not be sensible to send every sample. For example if a device is connected to the telemetry unit via a slow, expensive GPRS connection and the current sample rate was of the order of micro seconds. Not not only would the data transfer be very costly, but too slow to handle the volume of data. Therefore another interrupt is used whose timer is set separately to the sample rate timer. This is INT_TIMER2. Once this timer is triggered it sets a flag indicating that the latest sample should be sent to the connected device. The sample could be sent within the interrupt routine itself, however this should be executed as quickly as possible. INT_TIMER2 is disabled in its own interrupt routine and only enabled again once the sample has been sent.

Setting TIMER_2

Unlike TIMER_1, TIMER_2 is only an 8 bit counter. TIMER_2 is set differently to TIMER_1. TIMER_2 can increment every one, four or sixteen instruction cycles, resets once a specified value is reached, and the INT_TIMER2 interrupt routine is only run once the timer has reset a given number of times. The timer can reset at any value between 0 and 255 and can reset up to 16 times before the interrupt is triggered. For information on how the timer is setup for each SAMPLE_TIME please refer to D_FIRMWARE.C.

6.6 Main Program

D_FIRMWARE.C contains the main program which the PIC executes. This program utilises all the functions mentioned previously in this chapter.

When the program starts it initialises both the Bluetooth and GPRS modules. Once these modules are initialised it sits in a continuous loop processing any data received and sending samples to connected devices.

Incoming data is first processed by the appropriate gprs_process() or bluetooth_process() function. If it is determined to be data sent from a remote device then the function process_request_gprs() or process_request_bluetooth() is called. These functions determine what the request is and carry it out. Ideally they would have been combined into one function which accepted pointers to send and receive functions (GPRS or Bluetooth). This would allow the function to execute regardless of whether data was being sent and received via Bluetooth or GPRS. Unfortunately this could not be implemented as CCS does not support pointers to functions.

Live samples are sent when the `global_send_sample` flag is set. Samples are automatically taken via the `INT_TIMER1` interrupt routine as mentioned previously, and `global_send_sample` is set by the `INT_TIMER2` interrupt routine when it is time to send a sample.

If the signal reaches the critical level and no device is connected a warning SMS is sent to the phone number set by the remote device application. Only one SMS is sent regardless on the number of times the critical level is reached. This prevents the possibility of hundreds of warning message from being sent. Another SMS is only sent once a remote device has connected to the telemetry unit and disconnected again.

The flow of the program is shown in Figure 26.

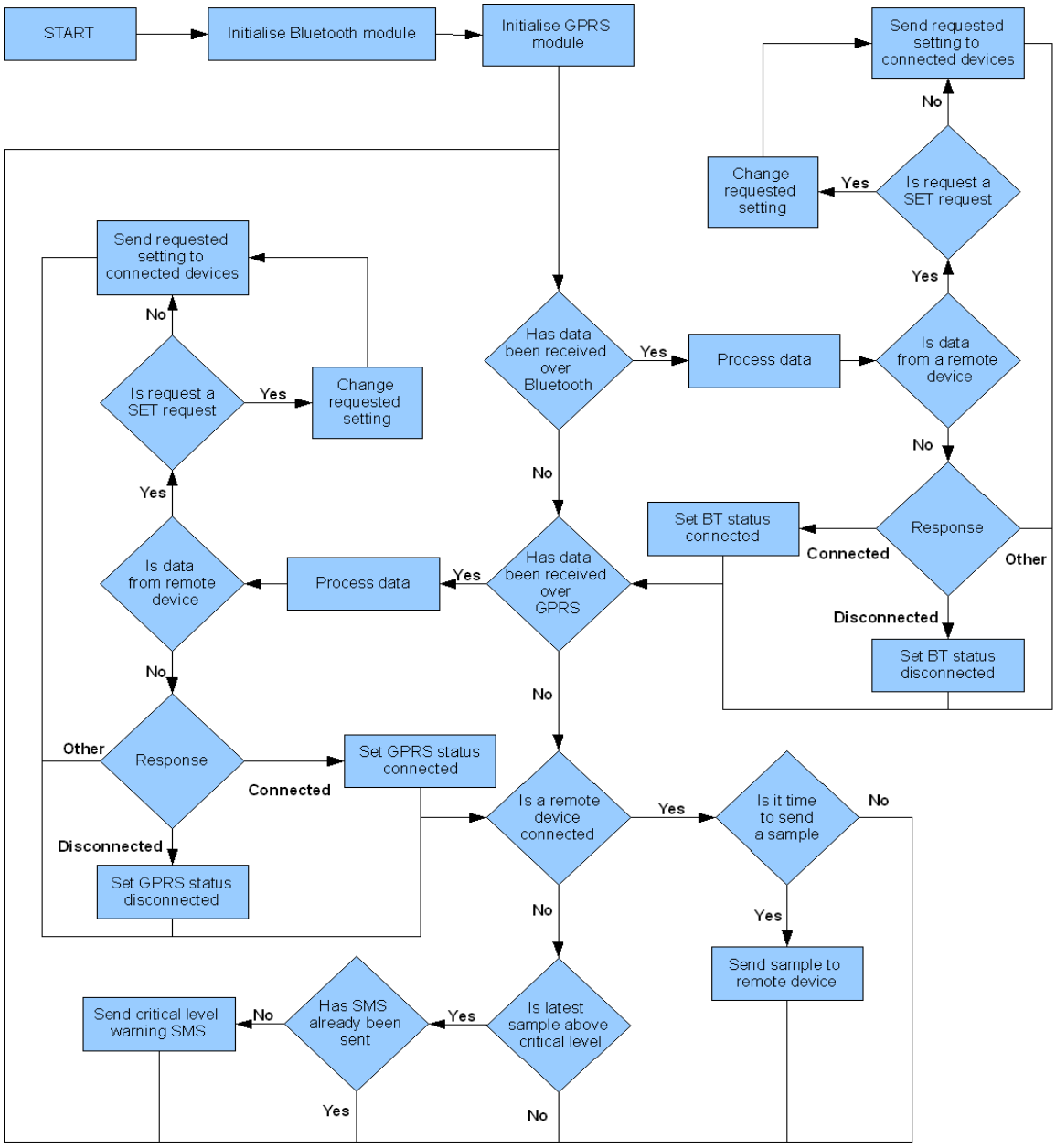


Figure 26: Main program flow.

7 Developing the Java Application

The following section describes the development of the Java application run by a remote device to control the telemetry unit was developed. All packages, interfaces, classes and methods detailed in this section were written for this project.

7.1 Java Application Design structure

When designing the Java application there were two main considerations:

- Portability
- Re-usability

Although Java programs can theoretically run on any platform there are still portability issues. Java contains some API which differs between platforms making it incompatible. This is evident in the user interface API. Java SE has powerful GUI tools such as `Swing`, whereas Java ME only has very basic GUI capabilities. A program written using the `Swing` API cannot be ported to Java ME and vice versa. Even if this were possible it would not necessarily be advised. The design of a mobile phone GUI has to take into account factors such as small screen size and limited user input whereas these are irrelevant for desktop applications.

For this reason programming the Java application was split into three separate packages:

- `UserInterface`
- `DataTransfer`
- `Connection`

This allows easy development of platform specific user interfaces without having to rewrite either the `Connection` or `DataTransfer` packages. A graphical user interface has been developed for standard mobile phones (see section 7.4) however a separate party could easily develop a user interface for a desktop PC without having to write any low level code as the `DataTransfer` and `Connection` packages provide API to handle this.

The `Connection` and `DataTransfer` packages handle connection creation and the transfer of data between the remote device and the telemetry unit. Separating these tasks enabled the `Connection` package to be developed generically, and all program specific code encapsulated within the `DataTransfer` package. This means that the `Connection` package can be utilised by other programs and programmers as its API is not specific to this application. This also allows for ease of updating or amending the `Connection` package if extra functionality is required. Zigbee connectivity could be enabled by simply adding an appropriate method to the `Connection` package without having to alter or understand the `DataTransfer` package.

The rest of this chapter goes through each of the three packages in more detail.

7.2 Connection Package

The `Connection` package simplifies the complexity of creating Internet and Bluetooth connections and sending SMS messages. All public methods of the `Connection` class return immediately and begin the connection process in a new thread. As a result these methods can be called freely without having to worry about blocking. Once the connection attempt is complete a callback is made to the `ConnectionstatusListener`,returning an instance of `StreamConnection` representing the connection made. The `ConnectionstatusListener` interface defines seven callback methods shown in Figure 27.

```
void bluetoothDeviceFound(String name)

void bluetoothDeviceSearchComplete()

void connectionEstablishedBluetooth(StreamConnection conn)

void connectionEstablishedGPRS(StreamConnection conn)

void connectionFailed(String error)

void smsSent()

void smsFailed()
```

Figure 27: ConnectionStatusListener interface callback routines

These callbacks are explained throughout this section. An instance of a class implementing the `ConnectionstatusListener` interface, to which all callbacks are made, is required by the `Connection` class constructor.

Creating a Bluetooth connection

There are two ways to create a Bluetooth connection. If the direct URL is already known then a call to `bluetoothConnectToUrl(..)` attempts to connect to the URL provided.

```
public boolean bluetoothConnectToUrl(String url)
```

However the URL is usually unknown. In this case `bluetoothSearchForDevices()` begins a search for all available Bluetooth devices.

```
public boolean bluetoothSearchForDevices()
```

Once a device has been discovered a callback is made to `bluetoothDeviceFound(..)` with the friendly name of the device.

```
public void bluetoothDeviceFound(String name)
```

Once all available devices have been discovered a callback is made to `bluetoothDeviceSearchComplete()`.

```
public void bluetoothDeviceSearchComplete()
```

A connection can then be made to one of the discovered devices by calling `bluetoothConnectToDevice(..)` with the friendly name of the device.

```
public boolean bluetoothConnectToDevice(String name)
```

If the connection is successful then a callback is made to `connectionEstablishedBluetooth(..)` providing a `StreamConnection` object for the connection.

```
void connectionEstablishedBluetooth(StreamConnection conn)
```

If the connection attempt is unsuccessful then an error text is returned through the

`connectionFailed(String error)` callback routine.

Creating an Internet connection

To create an Internet connection a URL and port number are required. The `gprsConnectToUrl(..)` method attempts to connect to the URL on the specified port.

```
public boolean gprsConnectToUrl(final String url, final int port)
```

If the connection is successful then a callback is made to `connectionEstablishedGPRS(..)` providing a `StreamConnection` object for the connection.

```
void connectionEstablishedGPRS(StreamConnection conn)
```

If the connection attempt is unsuccessful then an error text is returned through the `connectionFailed(String error)` callback routine.

Sending an SMS

To send an SMS the `sendSms(..)` method was developed which accepts the phone number and message as two `String` items.

```
public void sendSms(final String number, final String message)
```

A callback will subsequently be made to either `smsSent()` or `smsFailed()` depending upon the status of the SMS.

7.3 DataTransfer Package

This package contains two classes, `DataReceiver` and `DataWriter`. Together they handle all data transferred between the remote device and the telemetry unit. These classes require an `InputStream` or `OutputStream` instance which is opened from the `StreamConnection` returned once a successful connection is made by the `Connection` class.

There are two main purposes of this package. One of which is to hide the complexity of sending and receiving raw bytes behind simple methods and callbacks. The other is to ensure that the sending and receiving of data does not cause unexpected program blocks as this will result in a non-responsive user interface, freezing the entire application. Therefore all public methods in this package are non-blocking and return immediately.

Both `DataReciever` and `DataSender` extend the Java `Thread` class. This means that a call to `start()` begins the classes `run()` method in a new thread. It has been ensured that all methods of both classes are thread safe by using `synchronized` statements where required. For these classes to operate correctly a call to `start()` is made before calling any other methods. To end the thread created the `kill()` method was developed for each class.

DataReceiver

This class receives data sent to the device from the telemetry unit. All incoming data is handled in the separate thread to prevent blocking. This thread waits to receive data, processes it and then a callback to is made to a relevant `DataReceiverListener` method. The `DataReceiverListener` interface is shown in Figure 28.

```
void connectionConfirmed()

void gotSampleRate(int units, int largeMultiplier, int
smallMultiplier)

void gotDisplayRate(int units, int largeMultiplier, int
smallMultiplier)

void gotPhoneNumber(String phoneNumber)

void gotCritialLevel(int criticalLevel)

void gotSample(int sample)

void readTimeout()

void disconnected()
```

Figure 28: DataReceiverListener interface.

A class which implements this interface must be supplied in the `DataReceiver` constructor. All callbacks are made to the methods implemented by this class.

`DataReceiver` buffers all received bytes. These bytes are then analysed according to the command headers defined in `Header`. Once a valid header has been received `DataReceiver` waits for the remaining data needed to complete the command. A time-out begins. This ensures that any erroneous bytes received do not cause the thread to wait for data that is not being sent. This time-out starts a new thread which expires after a set time period. If all the data has not been received by the time this thread expires then `DataReceiver` ignores the current command and calls `readTimeout()`. If all the data is received in time then the time-out is cancelled, data processed, and if valid then the associated `DataReceiverListener` method is called.

Ideally the `DataReceiverListener` methods should be called in new threads so that `DataReceiver` can return to receiving data as soon as possible. However this would require all methods in any implementation of the `DataReceiverListener` interface to be thread safe. For simplicity this has only been implemented for the `gotSample()` method, demonstrating how this can be achieved.

The `DataReceiver` thread can be stopped at any point by calling the public `kill()` method. `DataReceiver` then closes the `InputStream` and kill the thread in which it is running.

DataSender

This class sends data to the telemetry unit. Calls to the `DataSender` public methods enqueue the data to be sent and return immediately. The separate thread sends the queued data.

The public methods of `DataSender` are shown in Figure 29.

```
public void setPhoneNumber(String number)

public void setCritialLevel(int criticalLevel)

public void setServerUrl(String serverUrl)

public void setSampleRate(int units, int largeMultiplier, int
smallMultiplier)

public void setDisplayRate(int units, int largeMultiplier, int
smallMultiplier)
```

```
public void requestSampleRate()

public void requestDisplayRate()

public void requestPhoneNumber()

public void requestCritialLevel()

public void requestServerUrl()
```

Figure 29: DataSender public methods.

These methods are split into to categories – set and request. The set methods are for changing the telemetry unit settings, and the request methods request for the telemetry unit to send the current value of the setting. The telemetry unit automatically sends back the value of a setting if an attempt is made to change it. These methods hide the complexity of sending correct headers and ensure that all data is formatted correctly. As mentioned previously they return immediately as data to send is queued and sent in the separate thread to ensure that no blocking occurs.

7.4 UserInterface Package

The `UserInterface` package provides a user interface for the `DataTransfer` and `Connection` packages. The `UserInterface` package developed for this project was written for mobile devices running Java ME. It hides `DataTransfer` and `DataConnection` methods behind aesthetically pleasing, easy to use graphical menus, icons text boxes and alerts.

J2ME Polish

Java ME provides several basic graphical user interface components. These include simple forms, lists, splash screens and alerts. After initial testing it was decided that these components were not adequate for what was required. Their appearance cannot be changed, menus cannot be renamed, navigation is awkward and overall they are very inflexible. A GUI could have been constructed using these components but it would have been far from ideal.

After research on the Internet, J2ME Polish was discovered. This is a tool suite which provides highly flexible user interface components. These components are similar to the ones provided by Java ME but are much more flexible, and have a fully customisable appearance which is defined in a separate CSS file. This allows programmers to create an interface and designers to make it visually appealing without having to change any code. The code must be compiled using the J2ME polish compiler which is provided with the suite.

Currently J2ME Polish is not supported by the Netbeans 'drag and drop' user interface design tool which allows users to visually develop user interfaces. Therefore all screens and navigation had to be hand coded.

J2ME Polish is available free for non-commercial use from [11].

Overview

The GUI class implements both the `ConnectionstatusListener` and `DataReceiverListener` interfaces. Connections are made using an instance of `Connection`. Data is sent using a `DataSender` instance and data is received through a `DataReceiver` instance. Its members include all the user interface components as well as the copies of the current settings. User interface screens are accessed via `getNameOfUIComponet()` methods which ensure that the UI components are constantly up to date displaying the latest settings received from the telemetry unit.

The remainder of this section describes some of the key methods of the GUI class.

Visual Appearance

The visual appearance of all the J2ME Polish user interface elements is defined by a CSS file. Different elements support different CSS attributes, however many attributes are supported by all elements. CSS styles can be define and used on many elements making it easy to experiment with different appearances. For detailed information on which CSS attributes are supported please see the J2ME Polish documentation which can be found at [11]. Among other attributes CSS was used to define:

- Element layout
- Text size / font / colour
- Background style and colour
- Border style and colour
- Hover effects
- Screen transitions
- Screen size
- Images

An example of the CSS code used to describe a section of the main menu is shown in Figure 30.

```
.mainScreen {

     padding: 2;
     padding-left: 10;
     padding-right: 10;

     layout: horizontal-expand | horizontal-center | vertical-center;
     view-type: dropping;
     screen-change-animation: fade;

     background {
           type: horizontal-stripes;
           first-top-color: brightBgColor;
           second-top-color: white;
           first-bottom-color: blue;
           second-bottom-color: black;
     }

}
```

Figure 30: Example CSS code

For a full list of CSS styles used please refer to polish.css in the Remote Device Application resources folder.

Keeping the user updated

To ensure that user is fully up to date with how the telemetry unit is operating a console window was developed. The console is a small window which automatically rises up from the bottom of the screen displaying information which is relevant to the user. If multiple items need to be displayed on the console then the user can either scroll through them or ignore them by closing the console. Typical information displayed on the console is connection status and confirmation of a changed setting. This is important if two remote devices are connected to the telemetry unit

as it keeps both users aware of changes the other is making.

Handling received data

When the `DataReceiver` instance (class member `read`) has received data, callbacks are made to the `GUI` class methods which implement the `DataReceiverListener` interface.

If the data received is the value of a telemetry unit setting then the member instance which represents that setting is updated and the console appears informing the user of this change.

If the data is the value of the latest sample then this value is displayed in the title bar of the main menu, and plotted on the visualiser if it is active. Updating the visualiser is a graphical rendering process which can take considerable processor time so is processed in a separate thread. During testing it was found that at very high display rates that samples were 'queuing up' to be rendered on the visualiser. This meant that the visualisation was no longer real-time. To prevent this if a sample is received and the visualiser is currently being rendered then the sample is ignored.

If the `DataReceiver` instance encounters an error (such as invalid data, or a read time-out) then a message window pops up informing the user of the error.

Sending data

When the user changes a setting using the various input forms this change is sent to the telemetry unit through the `DataSender` instance (`write`). The setting does not immediately change on the GUI just in case the change is unsuccessful. The GUI changes once the telemetry unit has confirmed the changed setting. The user is informed of this through the console.

Visualiser

Development began on a visualiser to display incoming samples in the style of an oscilloscope display before J2ME Polish was discovered. This visualiser was written using low level API, coding at a pixel level. The visualiser automatically adjusted to different device screen sizes, enabled zoom functions and resolution, time scale and axis labels could all be changed. However this visualiser was still in the early stages of development when J2ME Polish was discovered and time did not permit to continue development. This visualiser can still be accessed by clicking 'Full Screen' however it contains many bugs and much of the functionality is unavailable. The source code is available within the `UserInterface` package.

J2ME Polish provides a `ChartItem` class. This class is used to display numerical data in a visual form of bar charts, pie charts or line graphs. This class was created by J2ME Polish to display static data, however by constantly changing the data displayed by the chart appears like an oscilloscope display. When the device receives samples they are held in a shift register, with each incoming sample being entered at one end and the oldest sample being remove from the other.

The shift register is then plotted as a line graph. As more samples are received the previous samples shift through the register, with the graph replotted after each sample received. The graph appears to show the continuous flow of a signal. This solution is much simpler than the previous visualiser, however it is less flexible. Despite this it was decided that it was adequate and development of the other visualiser was put on hold.

If the signal goes above the critical level then the user is informed by exclamation marks in the title bar, and the displayed signal changes colour from blue to red.

7.5 Data flow diagram

Figure 31 shows the communication flow between the end user and the classes developed for this application. The thin black arrows represent methods being invoked.

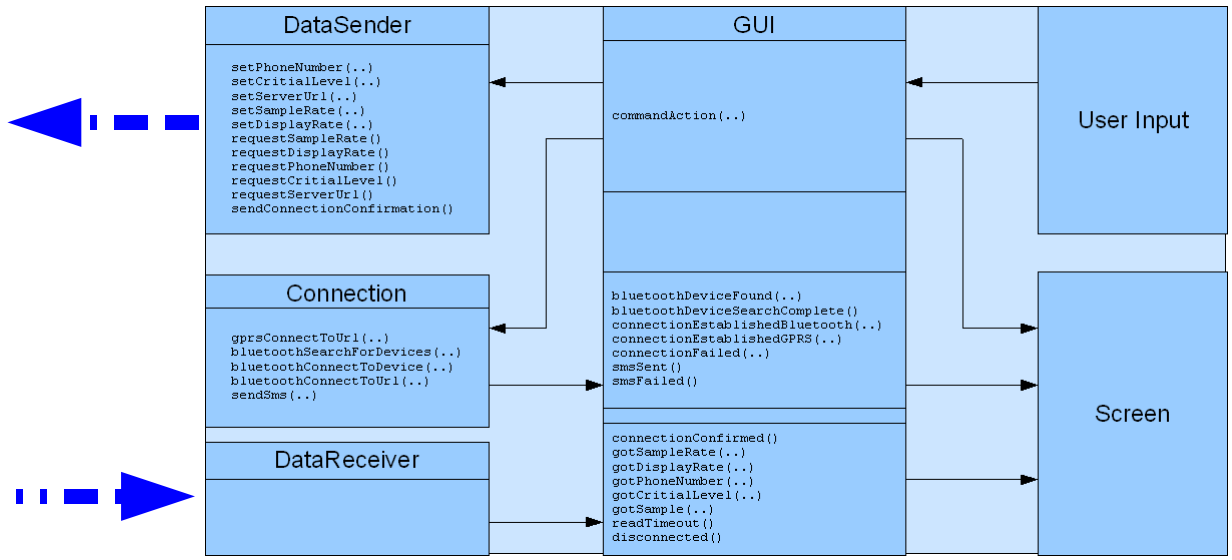


Figure 31: Flow of data in the remote device application

8 The Relay Server application

To enable access to the the telemetry over the Internet a relay server was developed. The relay server must be run on a desktop computer or laptop with a permanent internet connection. The relay server waits for two connections on a specified port and then relays data between the two. Once either connection closes the relay server closes the remaining connection, and waits for another two connections.

The relay server was also programmed in Java, and is contained in the `RelayServer` package.

8.1 RelayServer package

The `RelayServer` package contains the `SocketRelay` and `Sever` classes and `Serverstatus` interface. Each of these are described in more detail throughout this section.

SocketRelay

The `SocketRelay` class relays data between two sockets, provided in its constructor. It is a `Runnable` class as it must be run in its own thread. Two instances of `SocketRelay` are required to construct a fully functional relay because each instance relays data in a single direction – the output of one socket to the input of another.

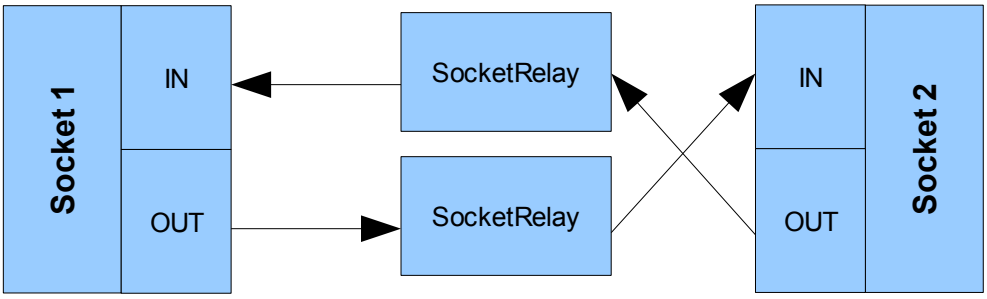


Figure 32: Diagram showing how the SocketRelay class relays data.

Each relay sits in an infinite loop relaying data until it detects that one of the sockets has closed. It then informs the `Server` instance (to which a reference was given in the `SocketRelay` constructor) that the relay has closed before shutting down all activity.

Server

The `Server` class is the main class in the `RelayServer` package. The `Server` class opens a server side socket on the designated port. It waits for connections on that port. Once one connection has been established it continues to listen for another connection. Once a second connection has been established a check is made to ensure the first connection is still alive before creating two instance of `SocketRelay` to relay the data. While data is being relayed no more connections are accepted. Once the relay has shut down all sockets and connections are closed before restarting the server, listening for the the next two connections. The public methods shown in Figure 33 were developed to control the server.

```
public void setPort(int portnum);

public void startServer();

public void stopServer();
```

```
public String getClient1IP();

public String getClient2IP();
```

Figure 33: Methods developed to control the server.

ServerStatus

`Serverstatus` is an interface defined to enable the `Server` class to send information updates about the current status of the server via call back routines. The call backs defined are shown in Figure 34.

```
    public void client1Conneted(boolean status);

    public void client2Conneted(boolean status);

    public void serverRunning(boolean status);

    public void relayActive(boolean status);

    public void writeBytesToConsole(byte[] bytes, int length);

    public void writeStringToConsole(String text);
```

Figure 34: Server status call back routines

These methods are called whenever there is a change to the status of the server. This enables monitoring of server activity and any data relayed over the server. The instance of a class which implements the `Serverstatus` interface is passed in the `Server` class constructor.

ServerVisual

`ServerVisual` is an example of a GUI for the `RelayServer`, implementing the `ServerVisual` interface. It enables the user to choose on which port the server should be run, as well as the option to start and stop the server. It displays the IP address of any connected clients, whether the relay is currently active as well as displaying any data that has been relayed.

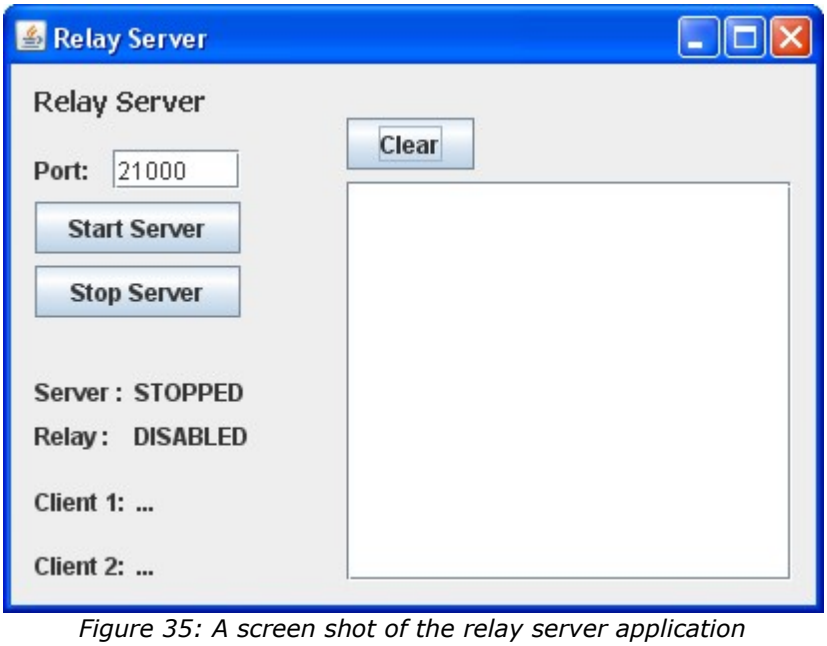


Figure 35: A screen shot of the relay server application

9 Telemetry System Data Flow

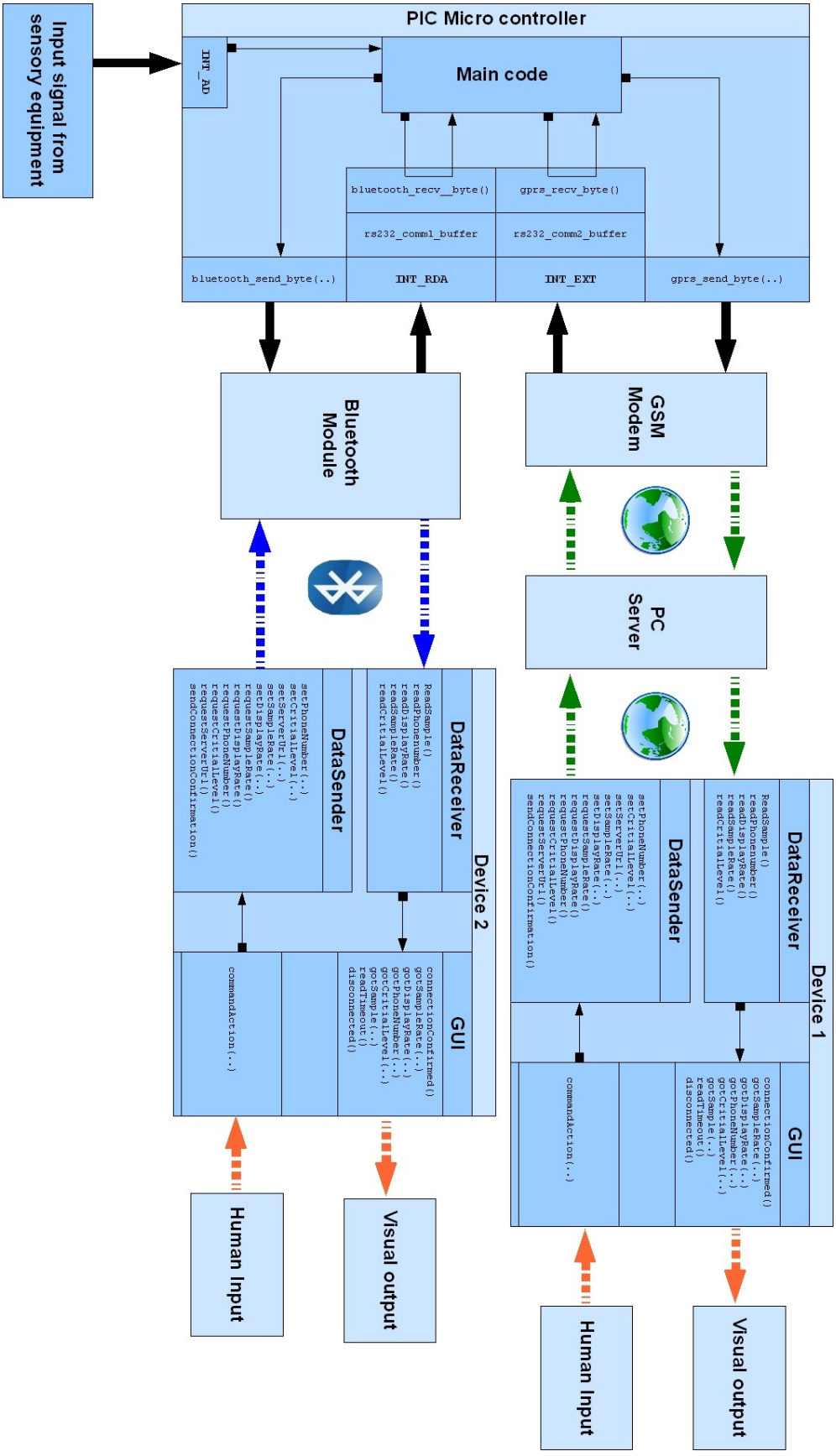


Fig 36: The overall flow of data through the entire telemetry system developed. Please refer to the relevant sections in chapters 7 and 8 for more information

10 Testing

Testing was split into three stages – telemetry unit testing, remote device application testing and overall system testing.

10.1 Telemetry Unit testing

As an object orientated approach was taken when programming the telemetry unit each class was tested individually to ensure its validity.

Data Structures

The first classes to be tested were the `buffer` and `circular_buffer` classes. This ensured that there were no memory leaks and data was stored and retrieved from the buffers correctly. Once these classes had been tested the RS232 interfaces were tested.

RS232 Interfaces

The RS232 interfaces were tested by sending bytes to and from a computer to the telemetry unit via an RS232 cable. These bytes were then displayed on the development board LCD and cross checked. Many different scenarios such as attempting to overflow the receive buffers were tested.

Bluetooth and GPRS

Once the operation of the RS232 interfaces had been confirmed the Bluetooth and GPRS functions were tested by sending commands to the Bluetooth module and GPRS modem respectively. The responses were then displayed on the development board LCD. The operation of the actual Bluetooth module and GPRS modem was first confirmed by connecting them to, and operating them from a computer.

Sample and Display timers

The timing of the sample and display timers was tested by changing the state of an output pin each time the interrupt was triggered. This output was then viewed on an oscilloscope to ensure the correct timing was being achieved.

10.2 Remote Device application testing

The remote device application was tested by creating a program that simulated the telemetry unit. This program was written in Java for a desktop computer. The program interpreted data and gave the same responses as way the telemetry unit would. This program had the ability to display exactly what data was being sent and received, allowing accurate debugging.

The program was used to test the remote device application when samples were being sent at very high rate. This was crucial in ensuring that the application could handle such speeds.

10.3 Overall system testing

Once both the telemetry unit and remote device application had been tested individually the overall system was tested. This proved very helpful in fine tuning the telemetry unit software. When two devices were connected and a high display rate was used it was found that the telemetry unit was prone to corrupting data and becoming stuck in infinite loops while sending /

receiving data.. This was prevented by refraining from using time consuming LCD statements., prioritising interrupts and disabling certain interrupts at appropriate times.

11 Telemetry System User Guide

This guide explains how to use the remote device Java application.

11.1 Getting Started

The telemetry unit control application is a Java application designed to control the telemetry unit. It enables live data analysis and full configuration of the telemetry unit.

Requirements

Any Bluetooth or GPRS enabled device supporting the Java ME CLDC 1.1 MIDP 2.0 profile or higher.

Installation

Download the application .jar and .jad files to the required device via the most convenient method, for example using Bluetooth or via a USB cable. Run the .jad file on the device and the application will be installed automatically. To run the application locate where it has been installed (usually in 'Games' or 'Applications') and click on the relevant icon.

Navigation

The application is easy to navigate, just use the default navigation keys on the device you are using. Options and menus are clearly displayed on screen, however more options for the current screen can often be accessed by clicking 'Menu'.

11.2 Using the application

Once the application has been launched the the main menu screen appears. This presents six options as shown in Figure 37.

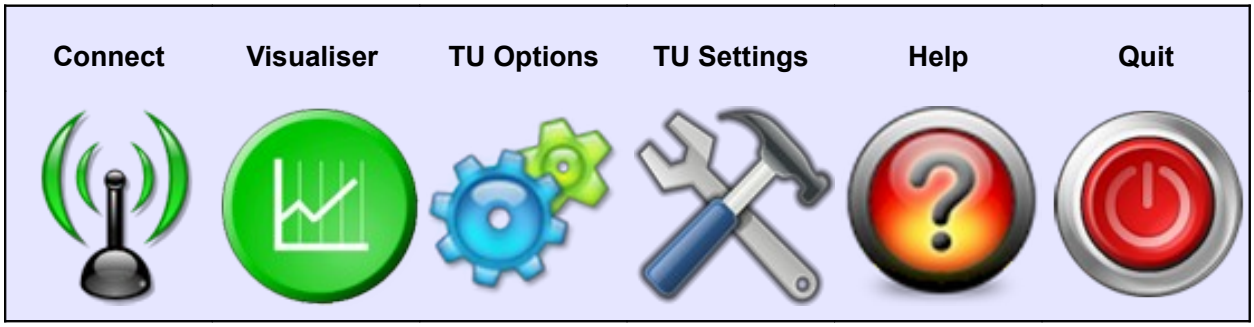


Figure 37: Main menu options

The first for options are explained in more detail throughout this section. The Help option displays a brief explanation of how to use this application and the Quit option exits the application.

Connection Menu (Connecting to the telemetry unit)

Once selected this takes you to the connection screen. From here you can choose how to connect to the telemetry unit. The connection types currently supported are Bluetooth and GPRS. The telemetry unit can support a single Bluetooth and a single GPRS connection simultaneously.

A connection demonstration is shown in Figure 38 on page 46.

Bluetooth

The application searches for all visible Bluetooth devices within range. Please be aware that the range of the telemetry unit is approximately 10m however obstructions such as walls may reduce this. Once all devices have been found (this will take a few seconds) a list is presented from which the telemetry unit can be selected. If the telemetry unit is not found then please check that you are within range of the telemetry unit, the device running the application has Bluetooth switched on and that this application has permission to access local connectivity. Once a successful connection has been made you are automatically re-directed to the main menu screen. If the connection fails an alert is displayed with the corresponding error message.

If possible connecting via Bluetooth is recommended over GPRS due to higher reliability, higher transfer speeds and free data transfer.

GPRS

A GPRS connection is made in two stages. An SMS is sent to tell the telemetry unit to connect to the server. Once this SMS has been sent the device itself connects to the server. Please be aware that there are costs involved in both sending and receiving data over GPRS and sending SMS messages. Please contact your service provider for pricing information. Once a successful connection has been made an you are automatically re-directed to the main menu screen. If the connection fails an alert is displayed with the corresponding error message. If the connection fails please ensure that:

- Both the device and telemetry unit have sufficient credit.
- Both the device and telemetry unit are connecting to the correct server address.
- The device has the correct phone number of the telemetry unit
- The server is currently running and not blocked by any firewalls
- The application has permission to send SMS messages and GPRS data

To change the server address to which the device connects and the phone number of the telemetry unit please scroll across to GPRS and then click on the 'Settings' option.

Visualiser

The visualiser plots samples sent by the telemetry unit. This can be used to analyse the signal sampled by the telemetry unit. The visualiser displays up to the last 50 samples received with a scale of 0-5V. The number of samples displayed at once can be altered through the 'Resolution' menu. This feature helps to maximise use of the screen. The rate which the telemetry unit samples the waveform and the rate at which samples are sent to the device can be changed via the 'Options' menu. At the bottom of the visualisation the duration of the signal displayed is shown.

If the signal rises above the critical level then the signal changes from blue to red. The critical level can be changed via the main menu 'Options' menu.

A demonstration of the visualiser can be found on page 47.

Telemetry Unit Options

The options menu allows you to change to the telemetry unit s sampling and data analysis options. The sample rate, display rate and critical level can all be changed. Once a setting has been changed it is not updated immediately on the application. The application waits for confirmation from the telemetry unit to confirm the change. A console window is shown once the changed setting has been confirmed.

An example of how to change a telemetry unit option is shown in Figure 41 on page 48.

Sample rate

The sample rate is how often the telemetry unit samples the incoming signal. This is changed by altering the three sliders presented. The first represents the time unit to be used, and the second two sliders are multiplied together to determine the number of this time unit to delay between each sample. There are over one hundred sample rates to choose from, ranging from 1us to 900 minutes.

Display rate

The display rate is how often the telemetry unit sends a sample to connected devices. Each time a sample is received its value is displayed in the main menu title bar and graphically in the visualiser if it is active. The display rate can be changed in the same manner as the sample rate.

Please note that due to the speed restrictions and expense of data transfer over GPRS if the display rate is set below 100ms a device connected via GPRS will not be sent any samples.

Critical Level

The critical level is the level which the incoming signal should never exceed, and if this happens immediate notification is required. The critical level can be set at any value between 0 – 256, with 0 representing 0V, 255 5V and 256 infinity (no critical level). If the critical level is reached you are notified by exclamation marks appearing on the main menu title bar. If no device is connected to the telemetry unit and the critical level is reached then a SMS message is sent by the telemetry unit to the phone number specified in the telemetry unit options.

Phone number

If the input signal reaches the critical level and no device is connected the telemetry unit will sends a warning SMS to this number.

Please note that only one SMS is sent, even if the input signal continues to stay above the critical level. This is to prevent the possibility of hundreds of SMS messages being sent. A SMS is only sent again once a device has successfully connected to and disconnected from the telemetry unit.

Remote Device Settings

The settings menu allows you to change the remote device connectivity settings.

An example of how to change a setting is shown in Figure 40 on page 47.

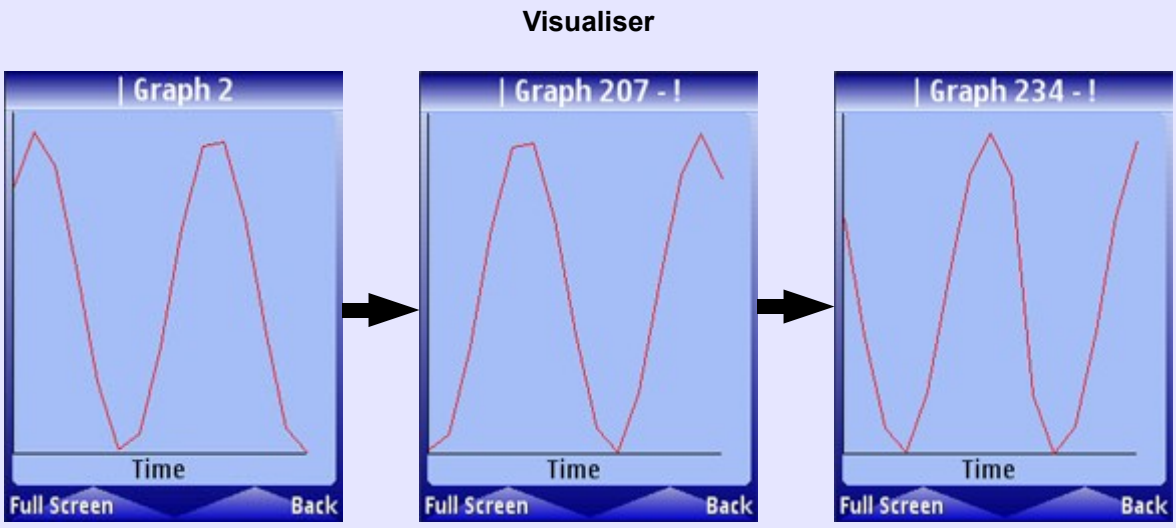
Phone number

This is the phone number of the telemetry unit. A SMS is sent to this number requesting the telemetry unit to connect to the relay server before the remote device connects to the relay server.

Server address

This is the address the remote device connects to when a GPRS connection is requested.

Figure 38: Bluetooth connection demonstration



Demonstration of the Visualiser displaying a signal. These screen shots were taken just a few samples apart.

Figure 39: Demonstration of the visualiser

Choose the 'Settings' option for the Main Menu

Choose the setting to change and press edit

Change the setting and press save

Figure 40: Demonstration of changing a device setting

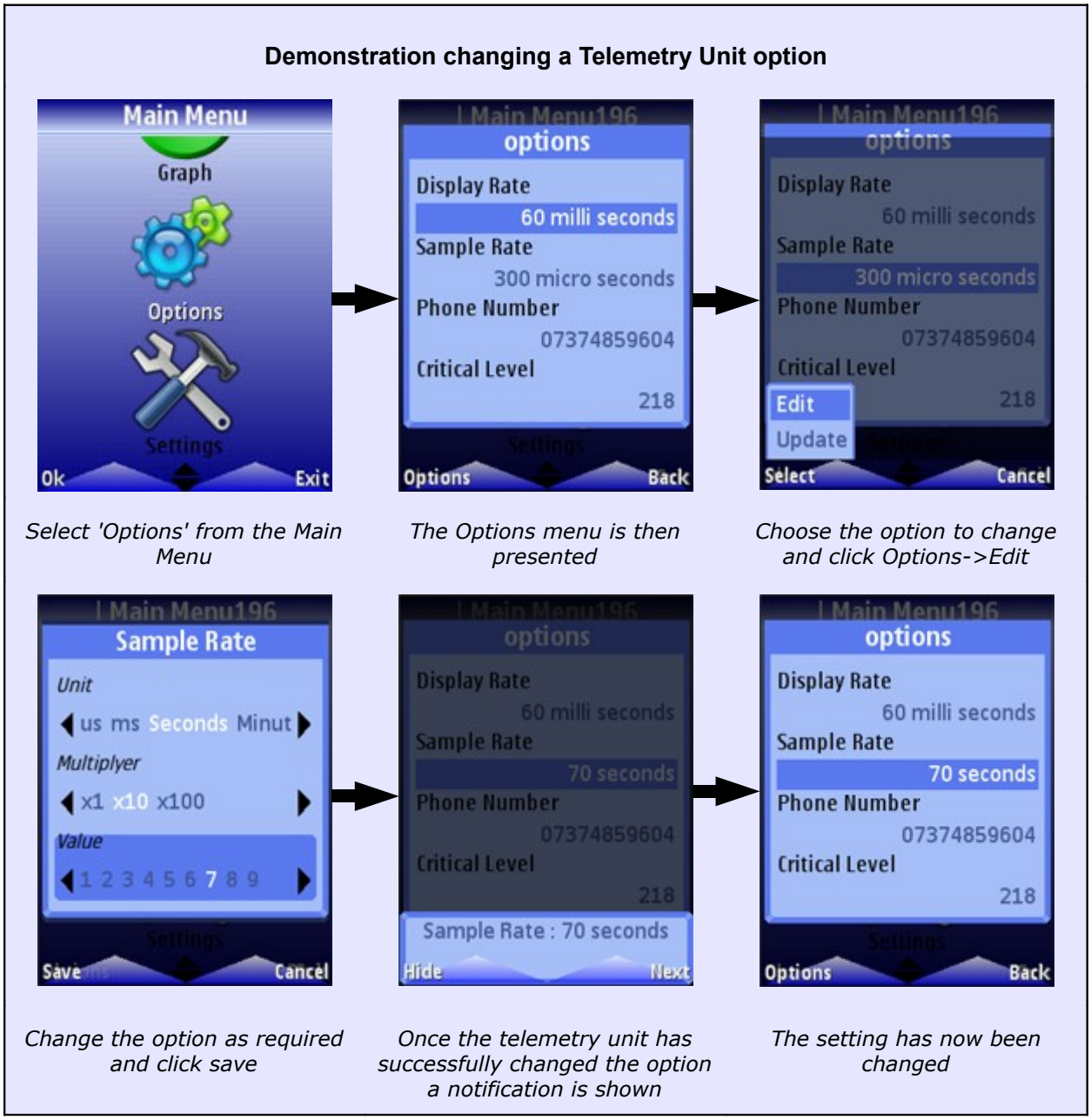


Figure 41: Demonstration of changing a telemetry unit option

12 Project Conclusion

The project has been a complete success. Development of the Wireless Telemetry System has exceeded all expectations and proved to be both an enjoyable and education experience.

12.1 Skills development

Whilst developing the wireless telemetry system many skills were acquired. They include:

- Java programming
- C programming
- Server programming
- Multi threading
- Interrupt handling
- Graphical user interface development
- CSS style sheet creation
- Researching and experimenting with new technologies

Technical skills and knowledge used throughout the project ensured the success of the system and provided innovative solutions to problems encountered; such as the creation of a relay server.

12.2 Wireless Telemetry System development

The system developed is accessible by any device with Bluetooth connectivity, an internet connection or GPRS connectivity from anywhere in the world. It is fully operational and very reliable. The remote device application is user friendly and has a graphical user interface which looks more professional than a lot of mobile Java applications currently available. The system is very flexible allows users full control of all settings.

Both the PIC software and Java application have been developed to allow extensions and improvements. The Java application is written so new user interfaces and connection technologies can be developed by third parties who are not required to understand the operation of the system at a low level.

The code provided is highly reusable and designed to be utilised by other programmers. The Java `Connection` package developed was used by a fellow student to create a Java application to transfer data over Bluetooth. This helped them to produce a successful final year project.

Through internet research no similar systems were found without spending a considerable amount of money. With the extra features described in the appendix this system could become a marketable product.

13 Appendix

13.1 Project Extension (Extra Features)

There are several features which could be added to improve the wireless telemetry system. These include:

- Attaching storage so samples can be saved for later analysis.
- Implementing a control interface so that a signal can be analysed and appropriate control action could then be applied by the remote device.
- Adding security to the system

These features could be implemented relatively easily. An SD card module could be attached to the telemetry unit allowing samples to be written to an SD card. This SD card could then be removed and data transferred to a computer, or the data could be sent via Bluetooth or GPRS if required.

A interface could be implemented on the telemetry unit to send control signals by simply raising the voltage on set pins or via RS232 communication. Commands could then be sent from the remote device and relayed on to control equipment by the telemetry unit. This would require adding functionality similar to the simple messenger application which passed messages from a remote device onto an LCD display.

13.2 Project Construction

Before the telemetry system can be used a simple PCB needs designing for the telemetry unit. Currently the PIC and Bluetooth module are powered via the development board. The PCB would also need to house a battery as well as the PIC and Bluetooth and GPRS modules. A connection input would also be required for the signal to be sampled.

14 References

[1] http://www.bluetooth.com/Bluetooth/Learn/Technology/Compare

[2] http://www.filesaveas.com/gprs.html

[3] http://www.microchip.com

[4] http://www.sena.com/products/industrial_bluetooth/esd.php

[5] http://www.roundsolutions.com/aarlogic/index.htm

[6] http://www.roundsolutions.com/gsm-terminal/index.htm

[7] http://www.sonyericsson.com

[8] http://www.ccsinfo.com

[9] http://notepad-plus.sourceforge.net/uk/site.htm

[10] http://www.netbeans.org

[11] http://www.j2mepolish.org

[12] http://developers.sun.com/mobility/apis/articles/bluetoothintro/index.html

[13] http://developers.sun.com/mobility/apis/articles/bluetoothintro/index.html

[14] http://www.dyndns.com

[15] http://en.wikipedia.org/wiki/Telemetry