University of Limerick

Department of Electronic and Computer Engineering

Final Year Project

Smart Phone Support for Bluetooth Body Sensors

Hugh O'Brien

ID: 0548111

obrien.hugh@gmail.com

B.E. Computer Engineering

Supervised by Dr. John Nelson

March 2009

Declaration

I hereby certify that this material is entirely my own work and has not been submitted to any other University or higher education institution, or for any other academic award in this University or other Universities. Where use has been made of the work of others it has been fully acknowledged and fully referenced.

Signed:

Date:

Abstract

This project sought to evaluate the potential of combining advances in cellular device technology with recent developments in the area of personal health monitoring. The advantages of this marriage of technologies are discussed and their applicability to the oncoming health care crisis is analysed. An analysis of the state of relevant wireless, sensing and portable computing technologies was performed. A proof of concept application was developed and the technological issues encountered in its production are documented.

The prototype is capable of real-time reporting of movement data to an off-the-shelf handset which can detect movements likely to be a fall and can then use cellular network to seek assistance for the victim and provide details of the incident. The prototype is also capable of monitoring electrocardiogram data.

Contents

1 Introduction **1**
1.1 Health Care Costs . . . 1
1.2 Health Sensors . . . 1
1.3 Portable Computing . . . 2
1.4 Combining Technologies . . . 2

2 Sensors **2**
2.1 TelosB . . . 3
2.2 The Mulle . . . 3
2.3 SHIMMER . . . 4
2.3.1 Micro-Controller . . . 4
2.3.2 Bluetooth . . . 5
2.3.3 802.15.4 . . . 5
2.3.4 Accelerometer . . . 5
2.3.5 Storage . . . 5
2.3.6 Power . . . 5
2.3.7 Input/Output . . . 5

3 Sensor Software **6**
3.1 Contiki . . . 6
3.2 TinyOS . . . 6

4 Portable Devices **8**
4.1 HTC Dream . . . 8
4.2 Nokia E51 . . . 8

5 Portable Device Software **9**
5.1 Java Platform, Micro Edition . . . 9
5.2 Android . . . 9
5.2.1 Development . . . 11
5.3 Symbian . . . 12
5.3.1 PyS60 . . . 13
5.4 Others . . . 13
5.4.1 Palm OS . . . 13
5.4.2 LiMo . . . 13
5.4.3 OpenMoko . . . 14
5.4.4 Windows Mobile . . . 14

6 Connectivity Options **15**
6.1 Cable 15
6.2 Wi-Fi 15
6.3 802.15.4 15
6.3.1 ZigBee 16
6.4 Bluetooth 16
6.4.1 Bluetooth Low Energy Technology 16

7 Prototype Specifications **17**
7.1 Sensor Specifications 17
7.1.1 Data 17
7.1.2 Connectivity 17
7.2 Portable Device Specifications 17
7.2.1 Connectivity 17
7.3 Software Specifications 17

8 Design **18**
8.1 Raw Data Processing 18
8.2 Sensor Software 18
8.3 Handset Software 18

9 Development Software **19**
9.0.1 Ubuntu 19
9.0.2 TinyOS 19
9.0.3 Build-system 20
9.0.4 VirtualBox 20
9.0.5 Windows XP 20
9.0.6 Putty 20
9.0.7 Python 21

10 Difficulties Encountered **21**
10.1 Bluetooth Support in TOS 2.x 21
10.2 Variable Sample Rate 21
10.3 Long Running Tasks 21
10.4 Fall Detection 22
10.5 Handset UI Design 22
10.6 PC Bluetooth Module 22

11 Results **22**
11.1 Description of Operation 22

12 Conclusions **23**

13 Acknowledgements **24**

A Appendix **25**

A.1 ThreeAxisToBluetoothM.nc 25

A.2 ThreeAxisToBluetooth.nc 32

A.3 Makefile 33

A.4 MovementGrapher.py 34

A.5 Demonstration 38

References **38**

Abbreviations

ADC Analogue to Digital Converter

API Application Programming Interface

ARM Advanced RISC Machine

ASCII American Standard Code for Information Interchange

BSD Berkeley Software Distribution

CPU Central Processing Unit

CVS Concurrent Versions System

EKG Electrocardiography

GDP Gross Domestic Product

GNU GNU's Not Unix

GPRS General Packet Radio Service

GPS Global Positioning System

GSM Global System for Mobile communications

HSDPA High-Speed Down-link Packet Access

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronic Engineers

IMEI International Mobile Equipment Identity

IP Internet Protocol

ISM Industrial, Scientific and Medical

J2ME Java (2) Platform, Micro Edition

JAR Java Archive

LAN Local Area Network

LED Light Emitting Diode

OS Operating System

PDA Personal Digital Assistant

PX Pixel

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RS-232 Recommended Standard 232

S60 Nokia Series 60 User Interface

SHIMMER Sensing Health with Intelligence, Modularity, Mobility and Experimental Re-usability

SMS Short Message Service

SPP Serial Port Profile

TCP Transmission Control Protocol

TOS TinyOS

TRIL Technology Research for Independent Living

UI User Interface

USART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

WLAN Wireless LAN

1 Introduction

1.1 Health Care Costs

Health care costs are on the rise globally, in the US they stood at 16% of GDP in 2007 and are expected to account for 25% by 2015[1].

Population ageing, the process whereby the median age of the people in a country rises, is occurring worldwide. In 1950, the global proportion of population aged 60 years or over was 8% today it is 11% and by 2050 it is expected to have risen to 22%[2].

The cost of health care for those above the age of 65 is estimated at between 2.8 and 3.5 times that of other groups[3].

The fertility rate in developed countries has dropped drastically since the 1970s. Ireland's 'Baby Boom' gives it one of the largest changes with the average children per woman dropping from 3.9 in 1970 to 2.0 in 2000[4]. By 2040 many of these people will have entered retirement; placing the increased cost of their health care on a substantially reduced workforce.

These statistics indicate the need for change in the area of health care for those over 60 years of age. There is much focus on the advancement of technology to aid in diagnosis and treatment of illnesses but comparatively little attention is paid to applying technology to the more mundane but equally significant health risks encountered by people in their own homes.

A common risk, and the target of the prototype application, is a fall. Hip related injuries are the leading cause of people being unable to care for themselves in their own homes and often necessitate permanent transfer to a health care facility. By reducing the incidences of falls it may be possible to enable people to care for themselves in their own homes, reducing the strain on the health care system.

1.2 Health Sensors

Health sensors are small, battery powered, wearable electronic devices that are designed to monitor some aspect of a person's biometrics. There are sensors available for movement, heart rate, blood pressure and other common health indicators.

Sensors represent a break in the trend of computer development as instead of engineering them to perform as many activities as possible a sensor is designed to do the absolute minimum required to accomplish its task. This benefits the power consumption of the device. As they are battery powered there is a finite amount of energy available to them and it must be used as efficiently as possible. Operational lifetimes in excess of one year from a single battery are a common target.

[1] Peter R. Orszag, US Congressional Budget Office Director; 'Growth in Health Care Costs'; Presentation to US Congress; 2008.

[2] UN Population division tables; World Population Ageing; 2007

[3] M. S. Marzouk; Ageing, Age-Specific Health Care Costs and the Future Health Care Burden in Canada; 1997

[4] UN population division tables; World Fertility Report; 2003

1.3 Portable Computing

Modern mobile phones possess a dizzying array of features and technology. They are mass produced to keep per unit costs low and have gained complete acceptance into the lives of most in the western world.

While it was historically difficult to access the capabilities of many of these devices, successive generations have exposed more and more functionality. Many devices have processor and memory capabilities comparable to desktop computers of yesteryear. The long range, high speed communication abilities along with low powered personal area network radios coupled with their inherent ubiquity make them the perfect target for research of this nature.

1.4 Combining Technologies

In this project I hope to demonstrate how these commodity consumer technologies can be used with relatively low cost health sensors to produce a basic but useful overview of the state of a persons health. While the data produced may not be as detailed as a checkup from a professional the 'always-on' nature of the technology may allow us to capture data that is otherwise unobtainable.[5]

Giving the patients themselves access to the data allows them to better care for their own health. If a physiotherapist were to recommend a minimum and maximum amount of activity per day then the patient could easily be informed as to their progress in reaching such goals without the therapist's intervention.

By transferring this type of basic health care from clinical premises to the patient's home significant strain can be removed from the health care system while improving the patient's quality of life.

The data produced by the system could be remotely analysed by professionals but also by expert systems which may be able to identify more subtle health issues than a standard check-up could.

By making use of real-time connectivity, emergency health conditions such as a heart attack can be quickly identified and will allow the system to immediately notify emergency services and provide potentially critical data about the conditions before the emergency.

Finally, the data of multiple patients with a specific illness could possibly be collected and compared to better understand the effects of the illness.

2 Sensors

Health sensors are a relatively new technology and as such, there is a limited selection of 'off-the-shelf' parts. The simplicity of the sensors[6] allows designers to build custom platforms tailor made for a particular project, unfortunately these designs are not readily available to the public.

[5] For example, studying the progression of a long-term disease
[6] Often fewer than ten active components

A number of standardised platforms have been created to encourage further research, these often have additional design considerations that make them easier for researchers to work with but are not designed for wide-spread adoption. They are best considered as prototyping targets, most of them are not specifically designed for use in health monitoring but as the technology is largely the same this is generally not an issue.

Several of these sensor platforms were considered for use within this project, their individual characteristics, merits and drawbacks are outlined below.

2.1 TelosB

The TelosB is a low cost sensor platform designed for research into low power radio technologies. It is manufactured by Crossbow technologies[23]. The design of the sensor is freely licenseable and as such is also available from moteiv under the name 'tmote sky'[33].

It uses a Texas Instruments MSP430 low powered micro-controller[7], a 250Kbps 802.15.4 radio[8] and antenna, 1MB of external flash storage, two expansion ports for connecting sensors, a standard USB port[9] for ease of programming, is powered by two AA batteries and is a supported platform for TinyOS and Contiki.

The device supports USB programming and communication by way of an FTDI[10] USB controller. The installation of a driver is necessary for the host to communicate with the sensor. Once installed, the device is accessible as a COM port under Windows or as a /dev/ character device[17] under UNIX like systems. Communication occurs through USART1 on the MSP430.

This device was immediately available to me and was the target platform for the majority of the TinyOS tutorials I worked through[18]. However the lack of on-board sensing and absence of a radio compatible with consumer handsets meant that it could not be used as the platform for the prototype.

2.2 The Mulle

The Mulle is a wireless sensor node with an embedded form factor, it weighs only two grams. It is manufactured by EISTEC AB[27]. The Mulle features a Renesas micro-controller with 31KB of RAM and 384KB of on chip memory, a Class 2 Bluetooth module or 802.15.4 transceiver, 4MB of flash memory, on-board temperature sensor and a three axis accelerometer (version dependant). It also features four analogue inputs for additional sensors.

Programming the Mulle requires a separately purchased expansion board however both the expansion board and the Mulle sensors themselves are competitively priced.

At the time of writing, The Mulle is an unsupported platform for Contiki but a port of TinyOS exists and is near completion.[20]

[7] See section 2.3.1

[8] A Chipcon CC2420, the standard 802.15.4 radio in wireless sensors

[9] Male, type 'A'

[10] Not an abbreviation, FTDI are a company that manufacture chips to translate RS-232 signals into USB signals

The Mulle is an attractive platform and would likely have been used had the SHIMMER not been available. However given the stark price difference between the two I would recommend future projects to strongly consider the Mulle[19].

2.3 SHIMMER

The SHIMMER is a wearable, battery powered wireless sensor designed by Ben Kuris and Steve Ayer while working at Intel Corporation. It is manufactured by Realtime Technologies[11] and is designed to monitor kinematic motion and physiological data.

It features a Texas Instruments MSP430 micro-controller, a Class 2 Bluetooth radio, 802.15.4 radio, 3 axis accelerometer, MicroSD slot (2GB Max) and TinyOS support.

The SHIMMER is also capable of measuring data from an additional gyroscopic sensor, EKG pads and any other arbitrary sensor through the analogue break-out board.

The Bluetooth radio is capable of operating as a serial port while abstracting away many of the complex issues of radio transmission.

USB programming is supported using an FTDI chip however the device must be connected to a separate docking station. Unlike other sensors, the SHIMMER includes a rechargeable battery, this provides cost benefits but also means the device must be removed from its sensor activities in order to recharge. The programming docking station allows the SHIMMER to recharge from the host system's USB interface[12]. A six port recharging station with mains adaptor is also provided.

In addition to the SHIMMER, the TRIL centre produce the BioMobius software package[29] that runs on a Windows system to assist clinical workers in producing meaningful data from studies such as gait analysis. There are specific TinyOS programs that may be uploaded to the sensor that are designed to work with BioMobius. While BioMobius was investigated, it was found not to be applicable to this project.

The SHIMMER was selected for use with this project as it had a powerful Bluetooth radio, wearable form factor, TinyOS support, rechargeable battery and there were several sensors available for use within the department. Some further technical details are given here.

2.3.1 Micro-Controller

The MSP430 family is a well known range of 16-bit micro-controllers from Texas Instruments designed for low cost, low power embedded applications. The model in use is the MSP430F1611. [30]

At 10KB, the model featured in the SHIMMER has the maximum amount of RAM in the family. It features 48KB of Flash memory which is programmable over RS-232. It also features eight independent 12-bit ADC channels. The maximum clock frequency is 8MHz.

These processors provide a variety of useful peripherals built-in, such as watchdog timers, DMA channels, SPI & I^2C connections and two USARTs. The device features its own high

[11] Ben Kuris and Steve Ayer have since begun working for Realtime Technologies[28].
[12] USB standard provides 500mA at 5V

frequency oscillator, removing the need for an externally calibrated crystal. In the device's lower power modes this oscillator can be disabled in which case the device is clocked from an external 32KHz[13] oscillator.

The MSP430 has surprisingly low power requirements, compared to similar micro-controllers from companies such as Atmel. When active at 1MHz the chip will draw only 330μ amps from a 2.2 volt supply. Standby mode draws a mere 1.1μ amps.[31]

2.3.2 Bluetooth

The Bluetooth module in use is a low profile Class 2 device with an on-chip aerial. It supports the full 79 channels of the Bluetooth specification with a maximum data rate of 2.2MBps. Current consumption (when active) is typically 50mA, making it the most draining device on the SHIMMER[32].

2.3.3 802.15.4

The SHIMMER uses the Chipcon CC2420; an 802.15.4 transceiver with ZigBee compatibility. It has a maximum data rate of 250Kbps, current consumption is 20mA on average. This chip is very popular amongst wireless sensors and as such is very well supported in TinyOS[25].

2.3.4 Accelerometer

The Accelerometer in use is Freescale's MMA7260Q low cost capacitive sensor. The chip features several selectable sensitivities, (1.5g, 2g, 4g, 6g) temperature compensation and consumes 500μ amps when active[26].

2.3.5 Storage

In addition to the 48KB of flash memory on the micro-controller the SHIMMER features a MicroSD card slot with addressing capabilities for up to 2GB of data[24].

2.3.6 Power

The SHIMMER includes a 250mAh rechargeable lithium-ion based battery with a nominal voltage of 3.7V. The battery is manufactured by UltraLife. As lithium-ion batteries are somewhat difficult to work with the SHIMMER also includes a dedicated battery management chip that ensures safe operation of the battery and provides information as to its charge level.

2.3.7 Input/Output

There are four coloured LEDs which are under developer control, these LEDs are invaluable for device debugging. There is also a reset button although this is only accessible if the sensor is not in its protective casing.

[13] A standard 'watch' crystal

3 Sensor Software

3.1 Contiki

Contiki is an operating system for embedded sensor devices. It has a focus on networked devices and provides a full TCP stack with support for IPv4 and IPv6.

Contiki is written in C and released under the BSD license. It is based around an event driven kernel upon which applications can be dynamically loaded and unloaded. This allows for the dynamic updating of applications on sensors already deployed in the field.

Contiki supports multi-threading and preemptive multitasking. Multitasking is a significant issue on memory constrained systems as the execution stack for each blocked process must be stored in memory. To alleviate this Contiki makes use of Protothreads[16], a form of lightweight thread.

There are a number of additional projects that extend the capabilities of a Contiki system, these include a low-power radio stack, a tiny web server for host interaction, power profiling mechanisms and an on-sensor file system.

Contiki has support for common micro-controllers such as the MSP430 and the AVR family from Atmel.

Contiki was a very attractive solution for this project, it supports a number of different sensors including the TelosB. However it does not at this time support the SHIMMER, for this reason TinyOS was used.

The stark differences in programming models between Contiki and TinyOS make for an interesting comparison of the developing trends in sensor computing. I would recommend future projects to explore the possibility of using Contiki where possible, although I found the level of documentation available for TinyOS to be superior.

3.2 TinyOS

TinyOS is an operating system for wireless sensor networks. A large number of existing sensor platforms are already supported, either by the TinyOS developers themselves or in the form on contributed code from the manufacturers of the sensors.

TinyOS began as a collaboration between Intel research and the university of Berkeley and as such is licensed under the Berkeley 'BSD' license. This allows the code to freely used and distributed in commercial systems.

TinyOS compiles the operating system and applications into a single binary, this allows the compiler to optimise function calls for improved code efficiency but prevents updates to the sensors while they are deployed as they must be individually reprogrammed.

TinyOS is non-blocking and has a single stack. This design requires that any operation that does not complete within some microseconds be dispatched to a queue to be executed at a later time. Operations such as I/O that traditionally make use of blocking must be handled asynchronously with the use of a call-back function.

Functions expected to take more than a few milliseconds to run are implemented as tasks.

Tasks are queued functions that are run when the CPU is next available. Posting a task is a very efficient operation and should take less than 80 cycles on average [13]. Tasks form the main 'workhorse' for data processing in TinyOS and their use is strongly encouraged.

In earlier versions of TinyOS there was a fixed queue size for tasks and the same task was allowed to be posted multiple times, this caused issues when the queue became full. TinyOS 2.x revised this by allocating a single slot in the queue for each task and allowing tasks to repost themselves. Tasks are not preempted. This non-standard design methodology can cause some confusion for developers new to the system, those accustomed to implementing threads in their applications may find it difficult to adjust.

TinyOS promotes code re-use by having programs makes use of functionality provided by components, these components can be implemented either in software or in hardware. The split-phase model of TinyOS (i.e. function dispatch with callback) allows for the abstraction of certain hardware facilities from the applications. e.g. if an application makes use of encrypted communications it may make an 'encryptPacket' call. If the hardware the application is currently running on has a dedicated encryption module the encryption will be carried out using it, if not, the packet is encrypted by the CPU (as a Task) when it is next idle. In both cases the same callback function is triggered once the encryption has completed allowing the application to operate without any awareness of the underlying hardware implementation, it need simply call upon the component related to encryption.

The entire operating system and its applications are written in nesC, a set of extensions to the C programming language designed to support the programming concepts of TinyOS. nesC programs are constructed by 'wiring' together components. Each component must declare the functions it implements and the the functions it calls. The components that provide the functions are 'wired' into the final program. Such wirings are specified in a separate source file.

When making use of a component you must define event handlers for each of the events generated by the components used. This allows for language enforced decoupling and promotes the creation of reusable and modular code. This is one of the core aspects of TinyOS/nesC.

The nesC compiler has the ability to optimise across object files, resulting in more efficient code than can be generated with standard C compilers, this is due in part to the bundling of the OS and the applications and is of great benefit in terms of sensor power efficiency. [13] [12].

TinyOS has an active user community, this is most apparent in their successful decision to document version 2 in a Wiki. TinyOS is being continually expanded through TinyOS Enhancement Proposals or 'TEP's. These provided detailed and easy to understand insights into the conceptual design of the OS. e.g. TEP 2, entitled 'Hardware Abstraction Architecture' explains in detail the three layers of hardware abstraction (Interface layer, Adaptation layer and Presentation layer) used in the OS. I would recommend reading all twenty of these documents before beginning TinyOS development. There are also a number of tutorials on the basic features of TinyOS that are well worth completing [18].

The community hosts annual conferences where interested users and companies meet to discuss upcoming developments. I attended the first European TinyOS Technology Exchange[10]

which included presentations and tutorials as well as demonstration sessions.

4 Portable Devices

4.1 HTC Dream

The 'HTC Dream' also known as the 'T-Mobile G1' or the 'Android Dev Phone 1' is the first smartphone designed to run the Android platform. It is designed and manufactured by HTC a Taiwan based smartphone manufacturer.

The phone features a 480x320px touchscreen display, five row QWERTY keyboard, dual-core ARM CPU, 192MB of RAM, 3.2Mpx camera and a MicroSD slot supporting up to 16GB of storage. The device also includes an accelerometer, digital compass and GPS receiver. Many connectivity options are provided, four GSM bands, two HSDPA bands, GPRS, Wi-Fi and Bluetooth Class 2[9].

The main draw of this particular device was its Android support. At the time of writing the handset is only available from T-Mobile in the US and UK although in December 2008 Google announced the availability of a developer model available from Google themselves at a fixed price. This model is only available to customers in 18 defined regions, of which Ireland is not included[8].

The difficulty in acquiring a G1 could be offset by the availability of an emulator for PCs. The emulator was specifically designed to allow developers to produce applications without having physical access to a device. This is further discussed in section 5.2.

4.2 Nokia E51

When it was realised that the G1 would not be available within the timeframe of this project I was forced to find a suitable replacement that was readily available and affordable.

The Nokia E51 is a general purpose phone that runs the Symbian Series60 operating system. This version of Symbian allows it to run binary applications without requiring an interpreter such as the Java virtual machine.

The device features a 240x320px colour screen, a four band GSM radio, HSDPA, GPRS, Bluetooth Class 2, Wi-Fi, 369MHz ARM 11 CPU, MicroSD slot, and 96MB of RAM[7].

Also of interest to this project is the availability of a text-to-speech engine built into the operating system. This engine is easily accessible to developers and opens up several options on how to make contact in case of an emergency.

This is a mid-range handset that represents the capabilities of most modern phones. The software produced for this device will work on any device that uses the Series60 platform however any binary applications are required to be signed.

5 Portable Device Software

There are a multitude of technologies in use in the realm of portable computing. Some, such as Palm, have been responsible for the first incarnations of the mobile computing ideal, while others, such as Android have entered more recently, in what they hope to be the blurring point of the distinction between a phones and a PDAs.

5.1 Java Platform, Micro Edition

When mobile operating systems were still directly tied to specific manufacturers or even to specific models the only way that an application could be developed for the mobile market would be if that application were specifically tailored to meet the needs of every device it hoped to run on. Even with the comparatively small selection of devices on the market at that time the concept was still infeasible.

In 1999 Sun Microsystems Laboratories launched 'Java 2 Micro Edition' commonly referred to as J2ME. This was a specific subset of the Java platform that provided a collection APIs common across all devices that implemented the standard. These APIs were chosen to aid in the development of applications for devices constrained by processing power as well as by small displays and lack of mouse or keyboard. Developers could produce and test applications on their personal computers by running the actual J2ME (written for the PC instead of a device) and be confident that the program would behave in the same manner on any device that provided the J2ME environment. For their part, device manufacturers need only implement one program, the Java virtual machine and their device could be marketed as supporting thousands of different applications.

J2ME was, and remains, widely implemented on almost all mobile devices of the last five years[14]. Running the J2ME VM can be quite taxing on a device's memory and as such it is not suitable for long-running background applications, it has however become a popular platform for mobile games as gaming applications tend to be stand-alone and have little interaction with anything but the user, the user also tends to not use other aspects of the device when using the game. As J2ME applications are constrained within the VM they are generally unable to access features of the device considered 'sensitive' often including telephony functionalities[15]. This was the primary reason J2ME was not a viable choice for this project[5].

5.2 Android

Android is an operating system for modern mobile devices. It is developed by Google and the Open Handset Alliance[16]. Android is based upon the Linux kernel and is available under an open-source license. Several handset manufacturers have announced plans to use the Android OS in upcoming devices although to date only the G1 has been released.

[14] It is interesting to note however that J2ME is not supported on the Android Platform

[15] It may be possible on certain handsets to grant extra privileges to the Java VM to allow it access such functionality but this possibility was not thoroughly explored

[16] A business alliance of near fifty firms tasked with developing open standards for portable devices[4].

Android applications are written in Java, however unlike the J2ME where Java applications are less privileged than native applications, Android Java applications form the base of the entire operating system. In a similar manner as Symbian applications, Android applications from 3rd parties are able to make use of the majority of the device's functionality such as accessing the camera or 3D acceleration features. This is in support of the Android philosophy that 'all applications are created equal'.

Following from that ideal, applications written for Android do not need to be signed before they can be run. Signing is the process in which the final version of an application is sent to the manufacturers of the target operating system. If the application meets their approval is it cryptographically signed and sent back to the author. It is commonplace for handsets and other devices (such as games consoles) to only run software that has been signed by the manufacturer. This is done to limit the availability of applications and to ensure that only programs that meet with the manufacturer's ideals are available. Most prominently, applications for the Apple iPhone are required to be approved by Apple before they can be used on the device, this contrasts with the PC world where there are no restrictions whatsoever imposed by the OS on what applications you are allowed to run.

One of the driving ideals behind Android is that applications make use of the functionality of the handset in new and novel ways. Having complete (though abstracted) access to the device's hardware allows for 'mashups', applications that combine data from different sources to produce a product more useful than the sum of its parts. Some examples of this include an alarm clock that is triggered not by time but by location (of interest to commuters who rest on the bus). Another example is combining data from the digital compass with GPS data and street level images (provided over the internet) to display travel directions on images of the roads as they appear to the user instead of on a traditional top-down view.

All standard applications that an Android device ships with such as a dialler or contact book are open source and can be modified or replaced by the end user. Android uses a version of SQLite as a central database for contacts, messages, saved locations, etc. As the database is provided by the OS it is accessible to any application that runs upon it, meaning that modifications made to your contact list by any application are visible to every other application that makes use of that list. This database can easily be synchronised with the user's online account.

There is a strong focus on online access with Android, with a large number of applications making use of Google's search, maps or email services. This has drawn some claims of vendor lock-in but the open source nature of the device allows it to access similar services provided by any other company.

Google also run the official 'Android App Store', a central software repository for Android applications. Both free and paid for applications are available for download direct to the handset. Developers who choose to sell their application on the App Store receive 70% of the fee paid by customers. This encourages small developers to produce applications that make novel use of the Android device's features and provides a simple, co-ordinated supply chain for their

product.

5.2.1 Development

While Android uses the Java programming language it does not use official Java VM from Sun Microsystems, instead it uses a custom designed VM named 'Dalvik' that has a register based architecture.[6]. This distinction prevents Android applications from being created using the existing Java compiler, to accommodate this; Google have provided a separate tool called 'dx' that converts a Java class and its dependencies into an archive similar to a JAR but suitable for use on an Android handset. The Dalvik VM was designed for use on memory constrained devices and as such has excluded certain features common in PC implementations of the Java VM. Most notable is the absence of 'Just-In-Time' compilation, a method that compiles frequently accessed methods into the platform's native instruction set on-the-fly to produce a speed boost at the expense of the memory required to store the new instructions. This was likely not implemented due to the memory constraints of most portable devices.

It is worth noting that as the Dalvik VM uses a different bytecode than Sun's VMs, it is not possible to run compiled J2ME applications on the Android platform. It may however be possible to implement the J2ME VM on the Dalvik VM given the recent open sourcing of Sun's Java platform, however with so many abstractions, the performance of such applications is likely to be poor.

There is an officially supported development environment provided by the Open Handset Alliance that runs on the majority of PC operating systems. Included in it is an Android emulator which is the actual binary system image of a generic Android handset compiled to the ARM instruction set. To run this image the QEMU[17] virtual machine emulator is used to emulate an ARM processor on the development machine. This allows developers to see exactly how their application will behave before uploading it to an actual device. It is possible to write and run an application that directly uses the ARM instruction set of the target device (bypassing Dalvik) but this development option is not supported by Google.

The official IDE is Eclipse. Eclipse is well known for its extensibility and Google have provided a plugin to assist with Android specific development. This allows the developer to compile, test, debug and package their application from within the one program. Eclipse is itself open source and written in Java.

Android provides libraries for 2D and 3D graphics based on OpenGL ES, a subset of OpenGL targeted for embedded systems. It also allows easy access to touchscreen data, GPS locations, and multimedia formats.

Notably missing from any version of Android is a fully developed Bluetooth stack. The existing system allows for the use of hands-free devices but is unable to support any other Bluetooth profiles. Work has been done in porting the open-source 'BlueZ' stack that is commonly seen in Linux distributions but this work is still at an early stage and requires modification of the handset's system image to implement. Only the Android Dev Phone allows the installation

[17] QEMU is a generic and open source machine emulator and virtualiser.

of unsigned system images (though all versions allow the use of unsigned applications) further restricting this option.

Unfortunately for the project this omission was not made clear until a lot of time had been spent researching Android and beginning to learn the Java programming language. After waiting a considerable amount of time to see if Bluetooth support was forthcoming it was decided that the project could not continue with Android and that a replacement system must be found with some haste.

Had Bluetooth support been present in the emulator the project could have proceeded without the handset (which was not available in this territory) however as the emulator runs the same binary file as the device this was not possible.

5.3 Symbian

Symbian is an operating system for mobile devices utilising ARM processors, it was originally owned by a consortium of handset companies[18] but has recently become wholly owned by Nokia[14]. Symbian allows for the installation of binary applications and opens up many aspects of the device's hardware to them e.g. it is possible to completely replace the device manufacturer's default program for handling SMS messages with one purchased from a 3rd party developer. This level of hardware access is not possible with the J2ME, although J2ME applications are fully supported on Symbian devices.

While Symbian describes the underlying operating system there are a number of different user interfaces that run atop it. Applications must be specifically written for each user interface, as each interface describes a different class of device. Sony Ericsson handsets with touchscreens use a UI known as UIQ, while many of the mid to high range Nokia handsets use an interface known as 'Series 60' or S60. There have been several editions of S60, the most recent of which is the fifth edition which is designed for upcoming touch screen devices. For this project S60 edition three, which was released in 2005; was used as it is the most commonly available edition. Binary installers are not compatible across different editions.

As of S60v3, all applications must be signed by a recognised signatory of the Symbian Foundation, this limits the amount of software that can be run on the device and gives considerable control to the foundation as to what applications may be distributed and sold. For development purposes, it is possible to apply for a key and certificate pair that will allow the developer to execute applications that are signed only by himself on one or two particular handsets whose IMEIs have been submitted and incorporated into the certificates. This allows the developer to test their application but prevents them from distributing it until it has been officially approved by the Symbian Foundation[1]. Due to the complications involved even in obtaining a developer's certificate, the demo application was not developed directly on the Symbian OS.

Symbian is in the process of becoming an open source project, possibly as a way of combating upcoming open source platforms such as OpenMoko or Android. The Symbian Foundation plan to release the entire OS as open source software during 2010.

[18] Nokia, Psion, Motorola and Ericsson

5.3.1 PyS60

Nokia maintain an interpreter for the Python programming language that is compatible with their S60 operating system. The interpreter is signed by Nokia and therefore can be installed on any S60 device. It allows for the creation of Python applications that do not need to be signed as they are not directly interacting with the OS. Python applications created in this way are somewhat limited in their functionality but are given more control over the device than applications running in the J2ME VM. e.g. through the S60 Python APIs it is trivial for a script to initiate a phone call.

The biggest penalty for using Python applications on a S60 device is the relative awkwardness involved in launching the Python interpreter and then directing it to the script to be executed. However the detailed call trace provided by the interpreter in the event of an application fault makes it a very useful platform for development.

Python is a dynamically typed, object oriented interpreted language. Applications written in Python may access the functionality of the underlying platform through exposed APIs. For S60 applications there are APIs for graphics, telephony, socket communications, text to speech functionality, audio recording and playback etc.

Being an interpreted language the data processing performance of Python is less than a native binary application and somewhat less than an application running within a VM; however the ease of development and detailed crash information make it an ideal choice for prototyping applications. S60 Python was chosen as the development language for this project. [11]

5.4 Others

5.4.1 Palm OS

Palm OS was one of the first operating systems targeting mobile devices, though in its case it was specifically made for Palm PDAs and related devices. Palm OS was first released in 1996 to power the then new 'pilot' range of PDAs. The rights to the software have since changed hands several times.

While initial versions were groundbreaking in providing mobile computing to a large group of people, subsequent versions have provided little in the way of new functionality to compete with the ever developing mobile phone market. This, coupled with the relatively small line of devices it runs on have led to its overshadowing by Symbian and other Smart Phone operating systems.

5.4.2 LiMo

The LiMo foundation is a consortium of several handset manufacturers including Motorola and Samsung who have joined together to develop the LiMo Platform, a mobile operating system based on Linux. There are over twenty handsets already running the LiMo platform.

I had not heard of LiMo until I began researching this project, they appear to have created a reliable operating system evident by its use on several real handsets, however I was unable to

find much information on how to develop for these applications which leads me to believe LiMo is more of a tool for device creators to use than a platform for software developers to target.

5.4.3 OpenMoko

OpenMoko is another project aiming at running open source software on modern handsets. The distinction between this and other Linux based platforms is that OpenMoko releases the hardware designs to their devices under an open license.

To date two handsets have been released, the Neo 1973 and the Neo FreeRunner. Both devices support the standard GSM bands and allow developers full access (within the extent of the law) to the underlying hardware. The FreeRunner compares favourably with more commercial handsets by providing Wi-Fi networking, two accelerometers and 3D graphics acceleration.

Both devices run OpenMoko Linux, a specialised distribution of Linux compiled for the devices but essentially containing the same components as found on PC Linux distributions such as X.Org[19], GTK[20] and Qt[21]. This allows most existing Linux applications to be ported to OpenMoko, an approach quite novel in the mobile computing space.

OpenMoko has relatively little industry backing but has garnered a large following among open source advocates. It is not tied to any particular carrier, nor does not impose or intend on imposing any form of mandatory application or operating system signing. It has been shown to work with kernels from other open source projects such as FreeBSD. The device was available for sale at two recent open source conferences I attended but can also be ordered from the manufacturer's website.

5.4.4 Windows Mobile

Windows Mobile, also known as Pocket PC is the second oldest smartphone operating system investigated, it was first released in 2000. Applications may be developed for Windows Mobile using Microsoft's .NET framework. Applications written in any of the .NET languages compile to a common bytecode that is run by the .NET virtual machine. Similarly to J2ME, the .NET libraries available on Windows Mobile are a reduced subset of the libraries available for the desktop edition. Programs written for one Windows Mobile device should work on any device running the same version of the OS.

As Windows Mobile devices tend to be at the higher end of the smart phone market it was not further researched as a platform as the one of the project goals was to build the prototype with relatively common devices.

[19] The standard windowing system.
[20] The GNOME graphics toolkit.
[21] The KDE graphics toolkit produced by the company Trolltech which coincidentally was recently purchased by Nokia.

6 Connectivity Options

As this project involves the pairing of two otherwise independent devices it is critical that both devices share a common method for communication. The possible methods are outlined below.

6.1 Cable

The simplest and most direct method of transferring data is to connect a cable from the sensor the the portable device. The devices could implement the SPI or I^2C protocol to allow for high bandwidth communication. Radio interference in the typically congested 2.4GHz band would not be an issue and it may be possible to power or recharge the sensor from the battery of the handset.

This solution, while conceptually the simplest might well the be hardest to implement as access to the I/O ports on most devices is quite restricted. Very few devices share a common connection method meaning that separate cables would need to be made for each new device the application was to run on.

The sensor could buffer its readings until such time as it was connected to the handset so data loss would not be an issue. This method of connection would rule out the possibility of taking action based on real-time information however and as such was not suitable for this project. I also believe that cables would be too much of an interference for the average user to tolerate.

6.2 Wi-Fi

Several of the latest handsets have built in IEEE 802.11g (Wi-Fi) capabilities. Wi-Fi is a high power, high data rate transmission system that uses expensive hardware. It is not suitable for use with mobile health sensors.

One area where Wi-Fi would be of use is if the sensor data gathered on the handset needed to be uploaded to a remote server on the Internet for analysis or for logging. Making use of the patient's Wi-Fi network would offer increased transmission speeds and reduced costs versus sending the data over the mobile carrier's network.

6.3 802.15.4

The IEEE standard 802.15.4 defines low powered short range wireless networks that operate in the 2.4GHz ISM band. The standard only defines the two lowest layers of the protocol stack, the physical layer (PHY) and the media access control (MAC) layer. It leaves definition of upper layers open to other standards.

802.15.4 has a nominal range of approximately ten metres and a maximum transfer rate of 250Kbps. Lower power modes are available with the trade off of lower transmission speeds.

This standard was specifically designed with low powered sensors in mind and is the most commonly supported connectivity option on the sensors reviewed. It would be a very attractive

technology to use for this project but as it is so tailored to wireless sensors it has seen little interest in any other devices.

6.3.1 ZigBee

ZigBee is one of several standards that define operations at the upper layers of the 802.15.4 radio protocol. Its specifics were not investigated.

6.4 Bluetooth

Bluetooth is a wireless protocol also operating in the 2.4GHz ISM band that is designed for low data rate communications over a relatively short distance. It has an average data rate of approximately 2Mbps and a range of up to 50m between two Class 2 devices.

Bluetooth is intended to remove the need for cables to connect the majority of consumer electronics. It is most commonly used to transfer data between two handsets that are in close proximity, to synchronise data between a handset and a PC, to replace cables in common PC peripherals such as keyboards and mice and to connect hands free devices to a handset.

Unlike 802.15.4, the Bluetooth protocol specifies the entire protocol stack and allows for devices to support extended features. These features are classified into Bluetooth profiles. These profiles define the type of data the device is capable of receiving. Common profiles are the 'Human Interface Device' profile for keyboards & mice, the 'Hands-Free Profile' for wireless headsets and the 'Serial Port Profile' for wireless emulation of an RS-232 serial port connection.

Bluetooth has well defined procedures for device pairing, allowing users to easily connect devices for the first time. It implements several forms of error correction chosen automatically based on the transmission properties. Encryption standards are also specified.

Bluetooth was chosen for use in this project as it is already widely supported on most handsets. Bluetooth is a mature technology and provides a complete solution for short range wireless data[3].

6.4.1 Bluetooth Low Energy Technology

An upcoming development of interest to this project is Bluetooth Low Energy. Previously known as 'Wibree' it is a form of low-powered Bluetooth specifically designed for embedded systems and wearable technologies. According to the press release

> [Bluetooth Low Energy Technology will be] a wireless technology with the same low power consumption and tremendous battery life as ZigBee, but able to communicate with the hundreds of millions of Bluetooth devices being shipped each year.

The technology is not currently available however future projects should strongly consider investigating its uses.

7 Prototype Specifications

7.1 Sensor Specifications

7.1.1 Data

There are many aspects of a person's health that can be monitored through electronic sensors. Many of these have specific requirements in how they are connected to the body, e.g. EKG pads must be coated in a special gel before being applied to the patient's skin. For ease of prototyping I chose instead to monitor the output of an accelerometer. This is a small integrated circuit which will vary its output voltage based on the acceleration forces acting upon it. This type of sensor is often used in gait analysis but is also suitable for fall detection. An accelerometer does not need to be placed anywhere in particular on the patient's body, in fact it will often suffice if the device is simply put in a clothes pocket. There are some advantages in the quality of data gathered if the sensor is close to the head but for this project this wasn't necessary.

The method in which accelerometer readings are taken is very similar to the way that additional sensors connected to the expansion board would be read. This allows the prototype to measure additional sources of information should additional sensors become available.

7.1.2 Connectivity

The sensor would communicate accelerometer data over a Bluetooth connection to the Portable Device. For ease of programming the data will be sent using the Serial Port Profile and will be formatted as the ASCII representation of the values returned by the accelerometer for the forces acting upon each of its axis.

7.2 Portable Device Specifications

7.2.1 Connectivity

The portable device will receive the data over Bluetooth and will have the capability to display it in a meaningful way.

7.3 Software Specifications

The software must be able to interact with the Bluetooth radio on the mobile device and be able to interpret and act on the data it receives. For testing purposes the software should be able to graph the data it receives. It should also be capable of using the telephony and data connections the mobile device provides.

This software will be developed as part of the project.

8 Design

8.1 Raw Data Processing

Some time was spent determining the balance of which device should process the data once it is acquired. This proved to be a complex issue as further processing on the sensor may lead to increased CPU activity but may result in less data being transmitted. Both these sensor components are significant users of power.

In order to ease development, and understanding that power efficiency was not the subject of the project, I decided to sample the accelerometers and send the values directly over the Bluetooth connection. It was not feasible to operate the sampler at its maximum frequency and send all the data over the radio as this caused a large load on the sensor. The handset also had difficulty operating at such frequencies as packet losses and retransmissions caused bursts of data which it was unable to fully process before the next samples arrived.

As a result of this, instead of attempting to operate the sensor and radio at their maximum frequency I decided that the sensor should perform some simple averaging of a number of samples and then transmit only the averaged values over the radio. This approach generates more meaningful data than single samples and is less intensive on both systems.

8.2 Sensor Software

The sensor software was designed to accomplish the following:

1. Initialise the Bluetooth radio and become available for connection.
2. Upon being connected to, initialise the accelerometer and sampler.
3. Sample the accelerometer at the defined rate and store the results.
4. Once a predefined number of results have been captured average them into a single result.
5. Broadcast the averaged result as an ASCII string to the connected device.
6. Repeat until connection is closed, whereupon the sampler is stopped and the sensor awaits a new connection.

8.3 Handset Software

The handset was required to search for and connect to sensors and receive the data they transmit. It was also required to perform some basic data processing such as normalisation. I decided that a simple graphing application would convey the sensor data more effectively than raw data so this was also implemented.

I chose to transmit ASCII data as it was easier to test the successful transmission from a PC console then it would be with binary data. The increased overhead is not particularly significant on packet sizes of this scale and may allow other device to interact with the data more easily. However a fully featured version of this prototype would most certainly make use of compressed

binary data in order to maximise the number of samples available and minimise the amount of data sent over the Bluetooth radio. The nature of the data leads to a high degree of similarities between readings which is exactly what is needed to obtain a high compression ratio.

In order to demonstrate the immediate usefulness of such a device I decided to implement a simple 'call out' feature to alert a nominated contact that the user of the device may be in distress. This feature dials a predefined number specified in the program, waits a fixed amount of time for the called party to answer the phone and then uses the text-to-speech engine present in S60v3 to read out the name and address of the victim and alert the contact as to what has happened. In future developments the handset may query the user as to whether they are in need of assistance and give them the opportunity to cancel the call out.

The handset process is as follows:

1. Search for available Bluetooth hosts.
2. Initiate a connection using the SPP.
3. Receive the ASCII data.
4. Process the data.
5. Display it on a graph.
6. Detect sudden changes in movement.
7. Alert the victim and a nominated contact.

9 Development Software

9.0.1 Ubuntu

The 8.04 version of the Ubuntu Linux distribution was chosen as the development system for the TinyOS part of this project. The TinyOS source code and dependant applications were easily installable through the package repositories provided by Stanford University. However these packages are only usable on systems that implement the apt package manager, these are often systems based upon the Debian Linux distribution of which Ubuntu is one of the more popular.

Any Linux distribution should be able to use TinyOS but the level of difficulty in installing the development environment will increase the further the user deviates from standard distributions.

9.0.2 TinyOS

The TinyOS development environment comes with several tools to assist with the development of TinyOS applications. These include necessities such as the nesC compiler as well as tools

to analyse code coverage[22], specific compilers for the various micro-controllers and the tools required to program them. There is also a large selection of example code that proved invaluable in developing this project.

While the software for the 2.x branch of TinyOS is easily installable through the provided apt[23] packages the source for the 1.x tree has not had a release since October 2003. In order to obtain the code it is necessary to use the source code management utility CVS to 'checkout' anonymous copies of the latest state of the code base. This is likely to be a stumbling block for anyone unfamiliar with SCM systems however once downloaded the source tree was found to contain everything needed to continue development.

I found it easier to install the entire TinyOS 2.x system from packages before attempting to download the 1.x tree as the packages also provided the much needed compiler tools that are common across TinyOS versions.

9.0.3 Build-system

The TinyOS development on Ubuntu was accomplished with the standard programming editor Vim and the GNU Make project compilation system. This simplified developing the SHIMMER application as only a small set of application specific compilation options needed to be set with the rest being imported from the TinyOS defaults. See section A.3.

9.0.4 VirtualBox

VirtualBox is a x86 virtualisation package developed by Sun Microsystems, it was used on Windows XP to allow a Ubuntu instance to run at the same time.

VirtualBox allowed the USB programming station of the SHIMMER to be passed to the Ubuntu guest system where it could be accessed as if it were natively connected.

9.0.5 Windows XP

Microsoft Windows XP was used for all other aspects of the project, it acted as a host OS to the virtualised Ubuntu instance. Windows provided a simple to use Bluetooth stack that conveniently mapped Bluetooth devices to serial (COM) ports so that terminal applications could view the output without being aware of how it arrived.

9.0.6 Putty

Putty is a terminal emulator for Windows, it was used to test and debug the output from the SHIMMER Bluetooth radio.

[22] The amount of code exercised by test functions
[23] A Linux package manager

9.0.7 Python

A Python interpreter was used under Windows in order to test certain aspects of the language before executing them on the PyS60 interpreter on the handset.

10 Difficulties Encountered

10.1 Bluetooth Support in TOS 2.x

When studying for this project I focused entirely on the 2.x release of TinyOS. This release was made in November 2006 and as of TinyOS 2.1 claimed full support for the SHIMMER. To my dismay I discovered that every aspect of the SHIMMER except for the Bluetooth radio had been implemented. This fact was not clear and was only discovered after many hours of believing I was calling it incorrectly. Eventually in desperation I began reading through the lower levels of TinyOS source and still could not find a single reference to Bluetooth. I believed perhaps that the name Bluetooth was abstracted away and encapsulated in the TinyOS ActiveMessage format but this was not correct. I later posted to the SHIMMER mailing list and was informed as to the lack of Bluetooth support in TinyOS 2.x. As a result development switched to TinyOS 1.x which has somewhat different methodologies in how tasks are handled, as well as different naming schemes for components which proved to be initially confusing. If any future developer wishes to use Bluetooth with TinyOS 2.x they must first port the RovingNetworksM.nc file to TinyOS 2.x. This task was beyond my skill level.

10.2 Variable Sample Rate

I had hoped to allow the Python application to send simple flow control commands back to the SHIMMER whereby it would adjust the frequency of the samples if the handset became overloaded, if the handset or the SHIMMER's battery was running low, or if the handset detected some suspicious results. Skeleton code for this feature exists in ThreeAxisToBluetoothM.nc (section A.2) but unfortunately due to time restrictions I was unable to get it working. I believe the issue is in how the SHIMMER interprets the data received over Bluetooth and may be solvable by further examination of the RovingNetworksM.nc file. That file defines how the Bluetooth module in the SHIMMER is accessed and what events it generates and is the only real reference for the Bluetooth functionality in the SHIMMER.

10.3 Long Running Tasks

It would seem that if a TinyOS task runs for more than several hundred milliseconds it begins to affect the performance of the radios. This caused some confusing network disconnections until the cause was determined.

One possible reason for this was mentioned by Phillip Levis [13] in his presentation; if the radio system posts tasks it may expect those tasks to be handled within a certain timeframe.

If the system is busy handling some long running tasks the task posted by the radio may miss its time window and cause some confusion for the network.

10.4 Fall Detection

Accurately interpreting the accelerometer data to determine when a fall has occurred is a difficult process. For simplicities sake the prototype code simply compares the levels between two adjacent samples and triggers the fall function if their difference is above a certain cut-off point. Such a system is not good enough for use as a proper fall detector.

10.5 Handset UI Design

Designing the layout of the graphs on the E51 was largely a trial and error process. The current implementation uses fixed pixel addresses and is completely unportable to any device that does not use a 320x240 screen. A better solution would be to express the pixel address as a percentage of the total available pixels. I would also advise any future developer to study the design principles of S60 applications so that the PyS60 application can act in a way that is consistent with the rest of the operating system.

10.6 PC Bluetooth Module

The USB Bluetooth module purchased for my PC required me to manually alter the device driver ini files to include support for its device ID. I would recommend against purchasing unbranded Bluetooth adaptors.

11 Results

After some difficulties the prototype software was successfully developed, it is able to connect to the SHIMMER, graph the data from the accelerometer on three independent graphs and initiate a distress call if there is a sudden spike in activity.

All software written for this project was developed entirely from freely-licensed software, which saved greatly on development costs.

Copies of the source code for this project are available in the Appendices.

11.1 Description of Operation

Once the SHIMMER has been programmed and the Python script is installed onto the handset the project can be demonstrated as follows:

1. Run the PyS60 interpreter on the handset.

2. Locate and run the script MovementGrapher.py.

3. Scan for and connect to the SHIMMER.

4. Connection will be confirmed on the handset and the SHIMMER will display changes in the LEDs.
5. Three different coloured graphs will begin to scroll on the handset screen. Move the SHIMMER to observe the detected motion.
6. Carefully apply a sharp acceleration to the SHIMMER, this should trigger the fall detection.
7. Observe the handset announce the detected fall and its intention to dial the listed contact.
8. After a short delay for the contact to answer the phone, an automated distress message is read out and repeated.
9. The handset returns to graphing movement.

A picture of this activity is given in section A.5.

12 Conclusions

This project exercised many different types of technology and encountered some non-obvious issues along the way.

- Health sensors are largely compatible with most mid to high end handsets. The technology can be adapted to produce a health monitoring system. It remains to be seen however if physicians will find the data produced to be useful.
- The SHIMMER is designed as a research platform, and as such is not quite as wearable as a more refined version could be, nevertheless it proved adequate for this project.
- The handset is quite a capable device, only basic features are made use of in this project but there exists the capability for complex analysis of the data received.
- The PyS60 application was surprisingly easy to work with, I would recommend future developers to prototype their application on this platform before spending the time required to implement it in Java or as a S60 application.
- The project does not make use of any of the security features provided by Bluetooth. When dealing with something as sensitive as medical information it is crucial that the data path be protected all the way from sensor to database.
- Throughput on the SHIMMER and MovementGrapher.py application could be increased through careful examination of the bottlenecks involved. I believe at this moment the 2D rendering on the handset is the slowest process and that the sample rate could be increased if used on a device with 2D acceleration.

- The fall detection system implemented here is far too simple to be viable in a real world environment. Any future application should make use of more intensive data analysis algorithms to correctly determine if a fall has in fact occurred [22].
- More recent, higher end handsets have accelerometers built into them, it may well be possible for fall detection as demonstrated here to be implemented solely through software on such devices. However the SHIMMER is still more versatile as it can accept other sensor inputs.
- The Bluetooth Special Interest Group has recently launched a Medical Device Profile as part of the Bluetooth specifications. This indicates the technology industry's willingness to accept e-health systems and should improve compatibility with handsets. Whether 802.15.4 becomes accepted or not remains to be seen.

13 Acknowledgements

- I would sincerely like to thank Dr. John Nelson for his continued support throughout all aspects of this project. His insight into the technology reaches far beyond the technical details and into the political reasoning of those who create these technologies. This project would never have developed without him.
- My thanks to my close friend David Dolphin for his knowledge of the quirks involved in academic papers and for his detailed experience in the convoluted world of smartphones.
- My thanks also to Phillip Levis for seemingly single-handedly documenting all aspects of TinyOS, and for producing an extremely detailed but surprisingly practical handbook for developers.

A Appendix

A.1 ThreeAxisToBluetoothM.nc

```
/*  Three Axis Accelerometer to Bluetooth Console for TinyOS 1.x
   Hugh O'Brien, March 2009. This code is released into the Public Domain.

   Samples are taken once every 20ms, once ten samples have been collected
   the results are averaged and sent over the Bluetooth radio, this results
   in an update frequency of 5Hz which should compromise between battery
   life and the loss of short lived readings.
*/

/* LED Legend:
   Green = Device is On
   Red = Bluetooth Connection Open
   Yellow On = DMA Buffer completed
   Yellow Off = Bluetooth Write Completed
   Orange  = DMA sample completed (toggles so blink rate = 1/2 sample rate)
   All On = Error State
*/

/* number of channels being sampled */
#define NUM_ADC_CHAN 3

/* take a sample every 'sample rate' miliseconds */
#define BASE_SAMPLE_RATE 50

/* how many samples to average before broadcast */
#define NUM_SAMPLES_TO_AVG 10

/*sum of the characters being sent */
#define CHAR_BUF_SIZE 16

/*     sensitivity is defined in the StdControl.init() function
   as 2G - it is itself a macro so could not be included here */


includes DMA;

module ThreeAxisToBluetoothM {
   provides {
      interface StdControl;
   }

   uses {
```

```
        interface Leds;
        interface Timer as samplingTimer;
        interface StdControl as BTStdControl;
        interface Bluetooth;
        interface StdControl as AccelStdControl;
        interface MMA7260_Accel as Accel;
        interface DMA as DMA0;
    }
}

implementation {

    /* define sprintf as a C type call */
    extern int sprintf(char *str, const char *format, ... )
    __attribute__ ((C));

    /* storage for DMA'd accelerometer data */
    uint16_t buf1[NUM_ADC_CHAN * NUM_SAMPLES_TO_AVG];
    uint16_t buf2[NUM_ADC_CHAN * NUM_SAMPLES_TO_AVG];

    /* keeps track of which buffer is being used by DMA */
    bool buf2_active = FALSE;

    /* keeps track of whether the alternate buffer is ready to use again */
    bool alt_buf_in_use = FALSE;

    /* counter for how many entries to the buffer have been made */
    uint8_t dma_blocks = 0;

    /* storage for ASCII output */
    uint8_t charbuf[CHAR_BUF_SIZE];

    /* on-the-fly modifiable sample rate */
    uint16_t sampleRate = BASE_SAMPLE_RATE;


    /* This is the first function called after the system 'boots' */

    command result_t StdControl.init() {
        call Leds.init();
        call AccelStdControl.init();
        call Accel.setSensitivity(RANGE_2_0G);
        call DMA0.ADCinit();
        call BTStdControl.init();

        /* Set up the flags for the ADC, code from BioMobius examples*/
```

```
    atomic{

        SET_FLAG(ADC12CTL1, ADC12DIV_7);

        /* sample and hold time 4 adc12clk cycles */
        SET_FLAG(ADC12CTL0, SHT0_0);

        /* set reference voltage to 2.5v */
        SET_FLAG(ADC12CTL0, REF2_5V);

        /* conversion start address */
        SET_FLAG(ADC12CTL1, CSTARTADD_0);

        SET_FLAG(ADC12MCTL0, INCH_5);   /* accel x */
        SET_FLAG(ADC12MCTL1, INCH_4);   /* accel y */
        SET_FLAG(ADC12MCTL2, INCH_3);   /* accel z */
        SET_FLAG(ADC12MCTL2, EOS);      /* end of sequence */
        SET_FLAG(ADC12MCTL0, SREF_1); /* Vref = Vref+ and Vr- */
        SET_FLAG(ADC12MCTL1, SREF_1); /* Vref = Vref+ and Vr- */
        SET_FLAG(ADC12MCTL2, SREF_1); /* Vref = Vref+ and Vr- */

        /* set up for three adc channels -> three adcmem regs -> three dma
        channels in round-robin */

        /* clear init defaults first */
        CLR_FLAG(ADC12CTL1, CONSEQ_2); /* clear repeat single channel */
        SET_FLAG(ADC12CTL1, CONSEQ_1); /* single sequence of channels */
    }

    /* initialise DMA, begin writing to buf1 */
    call DMA0.init();
    call DMA0.setSourceAddress((uint16_t)ADC12MEM0_);
    call DMA0.setDestinationAddress((uint16_t)buf1);
    call DMA0.setBlockSize(NUM_ADC_CHAN);

    /* these are flags specific to the MSP430 uC */
    DMA0CTL = DMADT_1 + DMADSTINCR_3 + DMASRCINCR_3;

    return SUCCESS;
}


/* this block executes after StdControl.init() */
command result_t StdControl.start() {
    call Leds.greenOn();
    call BTStdControl.start();
```

```
    return SUCCESS;
    /* System idles here until a BT connection is made */
}

/* this is only called in case of a (detected) system fault */
command result_t StdControl.stop() {
    call Leds.yellowOff();
    call samplingTimer.stop();
    call AccelStdControl.stop();
    call BTStdControl.stop();
    return SUCCESS;
}


/*     every time the sampling timer fires this is executed
    I suspect this might benefit from being atomic if the sample rate
    became very high but the examples do not show that thinking
    so I left it pre-emptable */

event result_t samplingTimer.fired() {
    call DMA0.beginTransfer();
    call DMA0.ADCbeginConversion();
    return SUCCESS;
  }


/*     simple task to remove the long running calls from
    Bluetooth.connectionMade, as it is called async it would otherwise
    run to completion */

task void postConnectionSetup() {
    call AccelStdControl.start();
    call samplingTimer.start(TIMER_REPEAT, sampleRate);
}

async event void Bluetooth.connectionMade(uint8_t status) {
    call Leds.redOn();
    post postConnectionSetup();
}

async event void Bluetooth.commandModeEnded() {
    /* we're required to handle this event but for this project we don't
    have to do anything meaningful */
}

/* long running async command is okay here as the system won't
```

```
    be doing anything else */
  async event void Bluetooth.connectionClosed(uint8_t reason) {
    call samplingTimer.stop();
    call Leds.redOff();
    call Leds.yellowOff();
    call Leds.orangeOn();
    call AccelStdControl.stop();
  }

  async event void Bluetooth.dataAvailable(uint8_t data) {
    /* code to allow the receiver device to modify the sample rate,
    disabled as it's buggy */

    /*call Leds.orangeToggle();
    atomic switch (data) {

      case 'S':
        sampleRate = BASE_SAMPLE_RATE / 2;
      case 'M':
        sampleRate = BASE_SAMPLE_RATE;
      case 'F':
        sampleRate = BASE_SAMPLE_RATE * 2;
    }

    call samplingTimer.stop();
    call samplingTimer.start(TIMER_REPEAT, sampleRate);*/
  }

  event void Bluetooth.writeDone() {
    call Leds.yellowOff();
  }


  /* this task is the CPU hotspot of my contributions however
    I may be calling OS provided functions that outweigh it */

  task void averageData() {
    uint16_t X=0,Y=0,Z=0;
    uint8_t i;

    /*    this doubled up code could be compacted by using
      a buf pointer but as space isn't currently an issue
      I left it verbose for clarity */

    /*if buf2 is being used by DMA, read from buf1 */
    if ( buf2_active) {
```

```
            for ( i=0; i <NUM_SAMPLES_TO_AVG ; i++ ) {
                X += buf1[NUM_ADC_CHAN * i ];
                Y += buf1[NUM_ADC_CHAN * i + 1 ];
                Z += buf1[NUM_ADC_CHAN * i + 2];
                }
            }
        else {
            for ( i=0; i <NUM_SAMPLES_TO_AVG ; i++ ) {
                X += buf2[NUM_ADC_CHAN * i ];
                Y += buf2[NUM_ADC_CHAN * i + 1 ];
                Z += buf2[NUM_ADC_CHAN * i + 2];
            }
        }

        /* we're finished with the buffer so allow DMA to use it */
        atomic alt_buf_in_use = FALSE;


        /* find the average values */
        X /= NUM_SAMPLES_TO_AVG;
        Y /= NUM_SAMPLES_TO_AVG;
        Z /= NUM_SAMPLES_TO_AVG;

        /* convert the values to fixed prescision strings */
        sprintf(charbuf, "%.4u %.4u %.4u\r\n",X,Y,Z);
        call Bluetooth.write(charbuf, CHAR_BUF_SIZE);

    }

    /*      This is called when the data from the ADC has
        been written to memory */

    async event void DMA0.transferComplete() {

        call Leds.orangeToggle();
        dma_blocks ++;

        /* move the DMA destination pointer along in the buffer */
        DMA0DA += (NUM_ADC_CHAN * 2);

        atomic {
            if (dma_blocks == NUM_SAMPLES_TO_AVG ) { /* if buffer is full */

                /*if the other buf is still in use, reuse the current one */
                if (alt_buf_in_use) {
                    if (buf2_active) {
```

```
                DMA0DA = (uint16_t) buf2;
                dma_blocks = 0;
            }
            else {
                DMA0DA = (uint16_t) buf1;
                dma_blocks =0;
            }
        }
        /* if it's not active then switch buffers */
        else {
            if (buf2_active) {
                DMA0DA = (uint16_t)buf1;
                buf2_active = FALSE;
                dma_blocks = 0;
            }
            else {
                DMA0DA = (uint16_t)buf2;
                buf2_active = TRUE;
                dma_blocks = 0;
                }

            atomic alt_buf_in_use = TRUE;
            post averageData();
        }
        call Leds.yellowOn();
        }
    }
  }


  async event void DMA0.ADCInterrupt(uint8_t regnum) {
    call StdControl.stop();
    call Leds.greenOn();
    call Leds.redOn();
    call Leds.orangeOn();
    call Leds.yellowOn();
  }
}
```

A.2 ThreeAxisToBluetooth.nc

```
/* Hugh O'Brien, March 2009 */

configuration ThreeAxisToBluetooth {
}

implementation {
  components Main,
  ThreeAxisToBluetoothM,
  RovingNetworksC,
  TimerC,
  LedsC,
  DMA_M,
  MMA7260_AccelM;

  Main.StdControl -> ThreeAxisToBluetoothM;
  Main.StdControl -> TimerC;

  ThreeAxisToBluetoothM.samplingTimer -> TimerC.Timer[unique("Timer")];
  ThreeAxisToBluetoothM.Leds -> LedsC;
  ThreeAxisToBluetoothM.DMA0 -> DMA_M.DMA[0];

  ThreeAxisToBluetoothM.BTStdControl -> RovingNetworksC;
  ThreeAxisToBluetoothM.Bluetooth -> RovingNetworksC;

  ThreeAxisToBluetoothM.AccelStdControl -> MMA7260_AccelM;
  ThreeAxisToBluetoothM.Accel -> MMA7260_AccelM;
}
```

A.3 Makefile

```
COMPONENT=ThreeAxisToBluetooth
PFLAGS=-DDEFAULT_BAUDRATE=115200
include ../Makerules
```

A.4 MovementGrapher.py

```
# Accelerometer Grapher and Fall Dector - Hugh O'Brien March 2009
#
#This is a script for PyS60 that opens a bluetooth serial connection
#to a pre-programmed SHIMMER sensor, The SHIMMER provides accelerometer
#data in the form "1111 1111 1111" where '1111' will be in the range
#of 0 -> 4400. The three values represent the data gathered
#from monitoring the three axis of the accelerometer.
#
#The script reduces the accuracy of these values in order to be able
#to graph them on a screen that is only 320x240px in size
#
#The script also monitors the difference between two subsequent
#readings in order to determine if a large movement has occured.
#This can be interpreted as a fall. A call is then placed to a
#pre-defined telephone number and the details of the victim are
#read out to the receiver.

import e32, appuifw, audio, telephone
#btsocket is the 'old' BT system, new version introduced in
#PyS60 1.9.1 is harder to work with.
import btsocket as socket

#a predefined BT MAC address can be set here to skip discovery process
target = ''

contact_name = "John Watson"
contact_number = "5550137"
victim_name = "Mr. Sherlock Holmes"
victim_address = "221 B. Baker Street. London"

sensitivity = 28

def fall():
    global app_lock, contact_name, contact_number, victim_name,\
    victim_address, data, prev

    audio.say("Dialling %s now" % contact_name)
    telephone.dial(contact_number)
    e32.ao_sleep(7) #7 sec delay for someone to answer
    for i in range(2, -1, -1):
        audio.say("This is an automated message. A fall has been detected.\
   Please assist %s at address %s. \
   This message will repeat %d more times" \
  % (victim_name, victim_address, i) )
```

```
    telephone.hang_up()
    data = ( 40, 40, 40 ) #reset values so as not to trigger again
    prev = data
    app_lock.signal() #unlock the main loop

def connect(): #this function sets up the BT socket connection
    global btsocket, target

    try:
         #socket params passed to the OS
        btsocket=socket.socket(socket.AF_BT,socket.SOCK_STREAM)

        if target == '': #if no target defined, begin OS discovery routine
            address,services = socket.bt_discover()
            target = (address, services.values()[0])

        btsocket.connect(target) #initiate connection and notify user
        appuifw.note(u"Connected to " + str(address), "info")

    except: #fail cleanly
        appuifw.note(u"Error connecting to device")
        btsocket.close()

def getData():    #this receives single characters over the bitstream
                #until it encounters a newline and carraige return it then
                #returns the characters it has buffered until that point

    global btsocket #use the globally defined socket
    buffer = "" #create an empty buffer
    rxChar = btsocket.recv(1) #receive 1 char over BT and save in rxChar

    #spin here until we get a 'real' char
    while (rxChar == '\n') or (rxChar == '\r'):
        rxChar = btsocket.recv(1)

    #as long as we receive 'real' chars buffer them
    while (rxChar != '\n') and (rxChar != '\r'):
        buffer += rxChar
        rxChar = btsocket.recv(1)

    return buffer #return the buffer contents


def graph_data(input):
```

```
#this function produces the graphs on the screen. the screen is
#landscape oriented with a resolution of 240x320. The constants seen
#here are used to define where on the screen the graphs should be drawn

global count, canvas, prev, data

#take the input string formated like "1111 1111 1111" and parse it
#to acquire 3 sets of chars and then interpret them as digits saving
#them to a list in this format: ( '1111', '1111', '1111' )
#the values are then divided by 60 as they will be in the range
#0 -> x -> 4400 as the screen is only 240px high. furthermore as there
#are three graphs being drawn each is confined to (240 / 3 )px of
#height. The divisor of 60 accommodates this at the cost of accuracy.

try:
    data = (\
    int(input[0:4]) / 60, \
    int(input[5:9]) / 60, \
    int(input[10:14]) / 60\
    )

#sane defaults if we receive a malformed reading
except ValueError:
    data = ( 36, 36, 36 )

#redraw the screen if there are more than 280 samples displayed.
if count > 280:
    reset()

#draw a line, with the X1 starting 10 points from the left and
#expanding right, Y1 being the previous value of Y2 (initially zero)
#plus a vertical offset so the graphs don't overlap each other, X2
#being one point right of X1 and Y2 one of the 3 XYZ readings plus
#the vertical offset. other options are purely aesthetic.
canvas.line(\
(count + 10, prev[0], count + 11, data[0] ), \
outline = 0xFF0000, width = 1)

canvas.line(\
(count + 10, prev[1] + 80, count + 11, data[1] + 80), \
outline = 0x00DD00, width = 1)

canvas.line(\
(count + 10, prev[2] + 160, count + 11, data[2] + 160), \
outline = 0x4444FF, width = 1)
```

```
    #increment counter - data should also be pushed into prev here
    #but this happens in the main loop for monitoring reasons
    count = count + 1

def reset(): # this function redraws the screen when it becomes full
    global count, canvas

    #reset the count and redraw a blank canvas
    count = 0
    canvas.rectangle((0, 0, 320, 240), fill = 0x000000)



#Main
data = ( 0, 0, 0 )
prev = (40, 40, 40) #initial zero values for 'previous values' of the data
canvas = appuifw.Canvas() #create a new Canvas object
appuifw.app.body = canvas
appuifw.app.screen = "full" #go 'fullscreen'
appuifw.app.orientation = "landscape" # draw in landscape orientation
appuifw.app.title = u"Activity Monitor" #name the program
app_lock = e32.Ao_lock() #locking system

connect() #open the BT socket
e32.ao_sleep(1) # sleep for 1 second in case of graphical slowness
reset() # initially reset the screen to draw the canvas

while 1: #loop the following code infinitely
    e32.reset_inactivity() #keep the screensaver away
    graph_data( getData() ) # poll the BT data passing it to the grapher.

    #test the movement level between the last two samples
    if ( (abs(data[0] - prev[0]) > sensitivity ) \
    or (abs(data[1] - prev[1]) > sensitivity ) \
    or (abs(data[2] - prev[2]) > sensitivity ) ):

        fall() #if too much, take action
        app_lock.wait() #pause this loop until fall() finishes
        e32.ao_sleep(1)
        reset()

    prev = data #move current data into previous data buffer
```

A.5 Demonstration

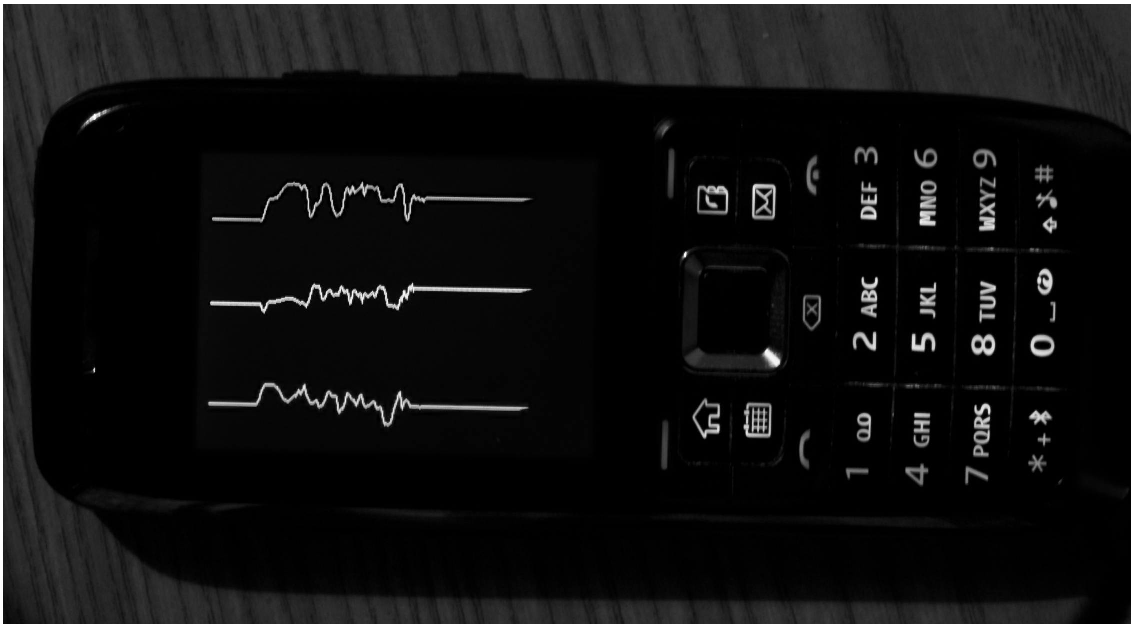

Figure 1: Plots of the three axis of movement generated by the SHIMMER

References

[1] The Symbian Foundation; How Do I get my Symbian OS Application Signed?; Accessed March 2009;
`https://www.symbiansigned.com/how_do_I_get_my_application_signed.pdf`

[2] Intel Corporation, TRIL Research Centre; SHIMMER Manual, Sensing Health with Intelligence, Modularity, Mobility and Experimental Re-usability;
Distributed by Intel 9th of May 2007

[3] Bluetooth Special Interest Group; Bluetooth.com - How It Works; Accessed March 2009;
`http://www.bluetooth.com/Bluetooth/Technology/Works/`

[4] Open Handset Alliance; Open Handset Alliance Overview;
`http://www.openhandsetalliance.com/oha_overview.html`

[5] Sun Microsystems; Java ME: the Most Ubiquitous Application Platform for Mobile Devices; Accessed March 2009;
`http://java.sun.com/javame/index.jsp`

[6] DalvikVM.com; Brief overview of the Dalvik virtual machine and its insights; Accessed March 2009;
`http://www.dalvikvm.com/`

[7] GSM Arena; Nokia E51 - Full Phone Specifications; Accessed March 2009; `http://www.gsmarena.com/nokia_e51-2106.php`

[8] Google, Open Handset Alliance; Developing on a Device, Android Developers; Accessed March 2009;
`http://developer.android.com/guide/developing/device.html`

[9] HTC Corporation; G1 Specifications; Accessed March 2009;
`http://www.htc.com/www/product/g1/specification.html`

[10] TinyOS Community; First European TinyOS Technology Exchange; 10th of February 2009, Cork, Ireland

[11] Jurgen Scheible, Ville Tuulos; Mobile Python: Rapid prototyping of applications on the mobile platform;
Published by Wiley, 2007; ISBN:978-0-470-51505-1

[12] Philip Levis; TinyOS Programming Handbook Revision 1.3;
Published On-line, 27th of October 2006;
`http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf`

[13] Phillip Levis; [Presentation] TinyOS - What the second generation holds;
17th of January 2007; Published by Stanford University;
Accessed March 2009; `http://www.youtube.com/watch?v=j6hRsue5b30`

[14] Symbian Software Ltd; Official Website, Accessed March 2009;
`http://www.symbian.com/about/index.asp`

[15] Swedish Institute of Compute Science, Adam Dunkels; Contiki Operating System; Accessed March 2009;
`http://www.sics.se/contiki/`

[16] Adam Dunkels, Oliver Schmidt, Thiemo Voigt; Using Protothreads for Sensor Node Programming

[17] The Linux Documentation Project; Character Device Drivers, Accessed March 2009;
`http://tldp.org/LDP/khg/HyperNews/get/devices/char.html`

[18] TinyOS Working Group; TinyOS Tutorials; Accessed December 2008;
`http://docs.tinyos.net/index.php/TinyOS_Tutorials`

[19] Jens Eliasson; Low-Power Design Methodologies for Embedded Internet Systems; Lulea University of Technology

[20] Laurynas Riliskis; TinyMulle, a port of TinyOS to The Mulle platform; Accessed March 2009;
`http://www.eistec.se/docs/wiki/index.php?title=TinyOS_Mulle`

[21] TRIL Centre; Accessed March 2009;
`http://www.trilcentre.org`

[22] Pepijn van de Ven, Alan Bourke, John Nelson, Gearid Laighin; A wearable wireless platform for fall and mobility monitoring; Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments.

[23] Crossbow Technology; TelosB DataSheet; Accessed March 2009;
`http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf`

[24] Intel DHeG Cambridge; SHIMMER Hardware Guide

[25] Chipcon AS; CC2420 SmartRF DataSheet;

[26] Freescale Semiconductor; MMA7260Q Technical Data

[27] Eistec AB; Product Website; Accessed March 2009;
`http://www.eistec.se/hardware.php`

[28] Steve Ayer; E-Mail response to the thread 'Accelerometer reading issues'; Shimmer-Users mailing list; 16th of April 2009;
`https://www.eecs.harvard.edu/mailman/private/shimmer-users/`

[29] Members of TRIL; The TRIL Centre BioMOBIUS Research Platform: An Open, Shareable Hardware and Software System;
`http://biomobius.trilcentre.org/images//biomobius_article.pdf`

[30] Texas Instruments; MSP430x1xx Family User's Guide

[31] Texas Instruments; MSP430x15x, MSP430x16x, MSP430x161x MIXED SIGNAL MICROCONTROLLER DATASHEET

[32] Mitsumi; Bluetooth Module, WML-C46 Class 2 DataSheet

[33] Sentilla Coroporation; tmote Sky DataSheet; Accessed March 2009;
`http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf`