Application of inertial sensing to handheld terminals

Gustaf Erikson
email: gustafe@home.se

Ericsson Radio Systems AB
164 80 Stockholm

Department of Teleinformatics
Electrum 204, 164 40 Kista

December 19, 2001

Abstract

We consider the potential for using inertial sensing in handheld terminals.

As an example, the implementation of an inertial sensing system in two dimensions for the research terminal Tifón at Ericsson Radio is discussed. The implementation includes hardware design, microcode software for an eight-bit micro-controller, and software for communication with the operating system of the terminal.

Acknowledgments

My heartfelt thanks to the following people:

- The Tifón team, at Ericsson Research: Peter Danielsson, Stefan Hellkvist, Johan Hedin, and Kourosh Pahlavan.
- My advisor, Gerald "Chip" Maguire Jr., at KTH IT, Kista, for not letting this ex-jobb die.
- David Sjögren for letting me grep *his* thesis report for layout ideas and help.
- Joanna, Hanna, and Leo, for being patient.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and background

This report is part of a Civ. Ing. project carried out at Ericsson Radio Systems AB and the department of Teleinformatics at the Royal Institute of Technology (KTH), Stockholm.

The purpose of this report is to detail the proposed implementation of an inertial navigation system for the Tifón, part of the Configurable Phone project at Ericsson Radio. It details the hardware implementation and microcontroller software for accessing the accelerometer data.

1.1 Background

The Tifón was part of a research project at Ericsson Radio called the Configurable Phone project. The configurable phone project's goal was to construct new communication devices which would be able to take advantage of both voice and data transmissions over a wireless network.

Data transmission (both over fixed lines and wireless) is becoming increasingly important. Indeed, it can be argued that the distinction between voice and data is increasingly artificial. However, from a user's point of view, there are certain differences in how one interacts with a voice channel and a data channel. With the advent of high-speed data transmission over wireless networks, a mobile phone will be used for both voice and data.

The traditional way of interacting with data (be it email, a terminal session to a remote computer, or the more graphical WWW interface) has been with a keyboard, mouse, and two-dimensional monitor. Much of today's data infrastructure is geared towards presenting information within this context. However, this puts handheld devices at a disadvantage. These devices are small, and getting smaller. They need to be light and have long battery lifetimes. New ways of interacting with data are interesting in this perspective,

as it will help users to be more productive in more situations.

1.1.1 The Tifón terminal

The Tifón tried to be as versatile as possible, under the constraints of being a handheld terminal. It had three configurations:

- Fully closed. The display is visible, and the user can make and answer calls, using the touch-sensitive display.
- Partly open. This exposes the keypad, which can also be used for text input. The display is also accessible in this configuration.
- Fully open. This reveals the touchpad, which can be used for handwriting recognition and screen manipulation. See Figure 1.1.

The terminal used an Intel StrongARM SA-110 processor, had 32 megabytes of random access memory (RAM), and 16 MB of read-only memory (ROM) which was used to store the operating system and applications.

The operating system was Linux, ported to the StrongARM [1]. The user interface was written in Java.

1.1.2 Input devices

There were four ways of interacting with the terminal:

- Touch-sensitive screen
- Keypad
- Touchpad
- Inertial sensing through the use of an accelerometer

The first three are familiar from portable and handheld computers. A good handwriting-recognition system [2] is included. The graphical user interface was planned to incorporate all the features necessary for accessing email, hypertext documents, calendar, address book and more.

The accelerometer is more novel. In fact, most of the devices using it are experimental ones. Section 1.2 gives some background on the use of accelerometers for navigation in handheld devices.

The basic idea is that the terminal can detect movement – or to be specific, acceleration. This of course includes gravity. So the user can move a cursor,

Figure 1.1: The Tifón in its fully open configuration.

for example, by moving the terminal laterally. Or she can scroll through text by tilting the terminal relative the gravity field.

Using the inertial sensing system in this way is perhaps practical; perhaps not. It is clear that the success of this feature of the user interface depends on a combination of good design on the part of the interface designer and a good device implementation. This report concentrates on the device implementation, as the (whole) Tifón system was not completed during the Civ. Ing. project.

1.2 Prior art

As computing devices become both smaller and more powerful, new ways of interacting with them are necessary if we are to realize their full potential. The traditional 2D-screen model, with overlapping windows, menus, and pointers, cannot scale to small screen sizes used in handheld personal digital assistants (PDAs) and mobile phones [3].

Proprioceptive devices, i.e. devices that are aware of their posture, movement and changes in position [4], offer a way for the user to use the movement of the device itself to interact with it. Accelerometers, capable of measuring lateral movement and gravity, and gyroscopes, capable of measuring rotation, can be used in such devices. Both of these sensors measure inertia. Other methods include using magnetometers, capable of measuring the Earth magnetic field, and various forms of sonar and infrared sensors.

The use of inertial components to measure gestures has increased due to the availability of cheap micro-electromechanical systems (MEMS). This technique permits the packaging of inertial sensors into components suitable for mounting on printed circuit boards. Verplaetse summarizes the techniques used in [4].

MEMS accelerometers are used extensively in the automotive industry for controlling airbags. Another application is the detection of movement of computer hard drives. Movement can be detected and the drive heads secured to prevent damage to the drive.

The research on using inertial devices to measure and recognize gestures has followed two main paths. One of them is using gestures as an interface to music generation and control. One such project is the Digital Baton [5], part of the Brain Opera project [6, 7] at the MIT Media Lab. The Baton uses accelerometers to measure large movements, and pressure sensors to measure finger pressure.

Another project is described in Sawada [8], which uses accelerometers to control a MIDI (Musical Instrument Digital Interface) by recognizing ges-

tures.

The other path is using inertial sensors as an interface to palmtop computers. Rekimoto [9] describes a palmtop system using tilts to navigate menus. The Itsy system from Compaq [10] uses accelerometers to detect tilting (for panning a digital photograph) and fanning (to zoom an image in and out).

The system described by Hinckley et al. [11] uses accelerometers to detect whether a PDA is held vertically or horizontally, and to detect, in conjunction with an infrared proximity sensor, whether the device is held in a position for recording a voice memo.

A cross-over between these two research areas is described in Marrin [12], where the advanced motion-sensing techniques developed for musical expression could be used for a general-purpose gestural interface.

A team a Xerox Parc has developed a handheld device called the Hikari[1] employing the interaction principles described in Fishkin et al. [13]

Benbasat [14] has developed a compact, wireless, six degree-of-freedom sensor package, together with supporting software and a scripting interface. This device promises to open new possibilities for researching gestural interfaces.

Although quite a lot of research has been done, no commercial implementations have reached the market yet.

The plan was to incorporate an Itsy-style of navigation in the Tifón, where tilting the device from side to side would choose between menu items. For this reason, a two-axis accelerometer was initially deemed sufficient.

1.3 Report overview

Chapter 2 details the hardware components used for the accelerometer system in the Tifón. This includes the accelerometer itself, the Programmable Interface Controller (PIC) used to process signals from the accelerometer, and the associated components.

Chapter 3 describes the software written for the PIC, used to process and analyze the signals from the accelerometer and present them to the main processor.

Chapter 4 describes the software written to test and verify the communication from the accelerometer under a development system using Linux.

Chapter 5 summarizes the results obtained from the project.

Chapter 6 details a test platform to be used for future work in 3D-motion sensing.

[1] An illustration of this device can be seen at `http://www.parc.xerox.com/csl/`.

Finally, Chapter 7 describe some of the possible applications of motions sensing in handheld devices, and summarizes the work done.

Chapter 2

Hardware implementation

This chapter will detail the hardware configuration used in the Tifón project.

2.1 Hardware overview

The following is a brief overview of the inertial systems parts which I have used and how they communicate with each other. Additional details will be presented in the following sections.

The accelerometer used is the ADXL202, by Analog Devices, Inc. [15]. It is a solid-state accelerometer with two measurement axes. The output from each axis is a pulse-width modulated (PWM) signal, where the width of the pulse is proportional to the acceleration experienced in the axis' direction.

The Tifón has a custom printed-circuit board (PCB) containing the CPU, memory and other hardware necessary for the computer. The accelerometer is mounted on this board at an angle of around 45 degrees to the main axis of the terminal. This gives the possibility of combining the output of both accelerometers during most movements.

The accelerometer is connected to the CPU of the terminal via a micro-controller (also called a programmable integrated circuit or PIC). The device used is a Microchip 16C622 [16]. The micro-controller is responsible for decoding the PWM signals from the accelerometer and converting them into numbers describing the acceleration. The micro-controller also takes care of the touchpad signals.

The PIC communicates with the StrongARM CPU by two wires. One of these is used for timing signals, the other for data. The protocol used is a simple serial one partly derived from previous work on the touchpad [2]. The data is sent in packets, with different headers for touchpad and accelerometer data. The protocol is described in Section 3.4.1.

2.2 Component integration

The wiring for the accelerometer and its connection to the PIC is easily separated from the rest of the system. These subsystems are shown in Figures 2.1 and 2.2.

The accelerometer subsystem consists of two chips: the accelerometer itself (described in Section 2.3), and the Microchip PIC 16C622 (described in Section 2.4).

2.2.1 Power requirements

The Tifón's power supply is a battery with a voltage of 3.0 V. The ADXL202 requires a supply current of 0.6 mA [15]. Obviously, power consumption is a critical part of the design of a handheld device. Whether the accelerometer will be a significant power drain is not easy to say, as all the parts of the Tifón were not ready at the time of writing. The Tifón's specifications dictate a standby time on par with modern mobile telephones (i.e. between 100 and 200 hours), but the operating power requirements are higher. However, it is probably safe to say that the accelerometer is one of the least power-hungry parts in the terminal. The screen and telephone components far outweigh it in this regard.

2.3 The accelerometer

The ADXL202 accelerometer [15] is a solid-state accelerometer in an integrated-circuit package. The device is delivered as a 8-lead LCC (leadless chip carrier). It's dimensions are $10 \times 9.9 \times 5.5mm$. It fits (barely) into the Tifón. It uses polysilicon springs to suspend a surface machined polysilicon structure over a silicon wafer. A differential capacitor measures the deflection of the structure. This is translated by signal conditioning circuitry into a duty-cycle modulated signal, which is easy to decoded by a timer on a micro-controller.

The accelerometer measures acceleration in two axes. This means that acceleration perpendicular to the plane defined by the axes cannot be detected. To measure acceleration in three dimensions, additional two-axis accelerometers, mounted at an angle to each other, can be used.

Another solution is to use a three-axis accelerometer, for example the Crossbow CXL02TG3[1] , or the Measurement Specialities ACH-04-08-05[2].

[1] `http://www.xbow.com/pdf/TG%20datasheet.pdf`

[2] `http://www.msiusa.com/805spec3.pdf`

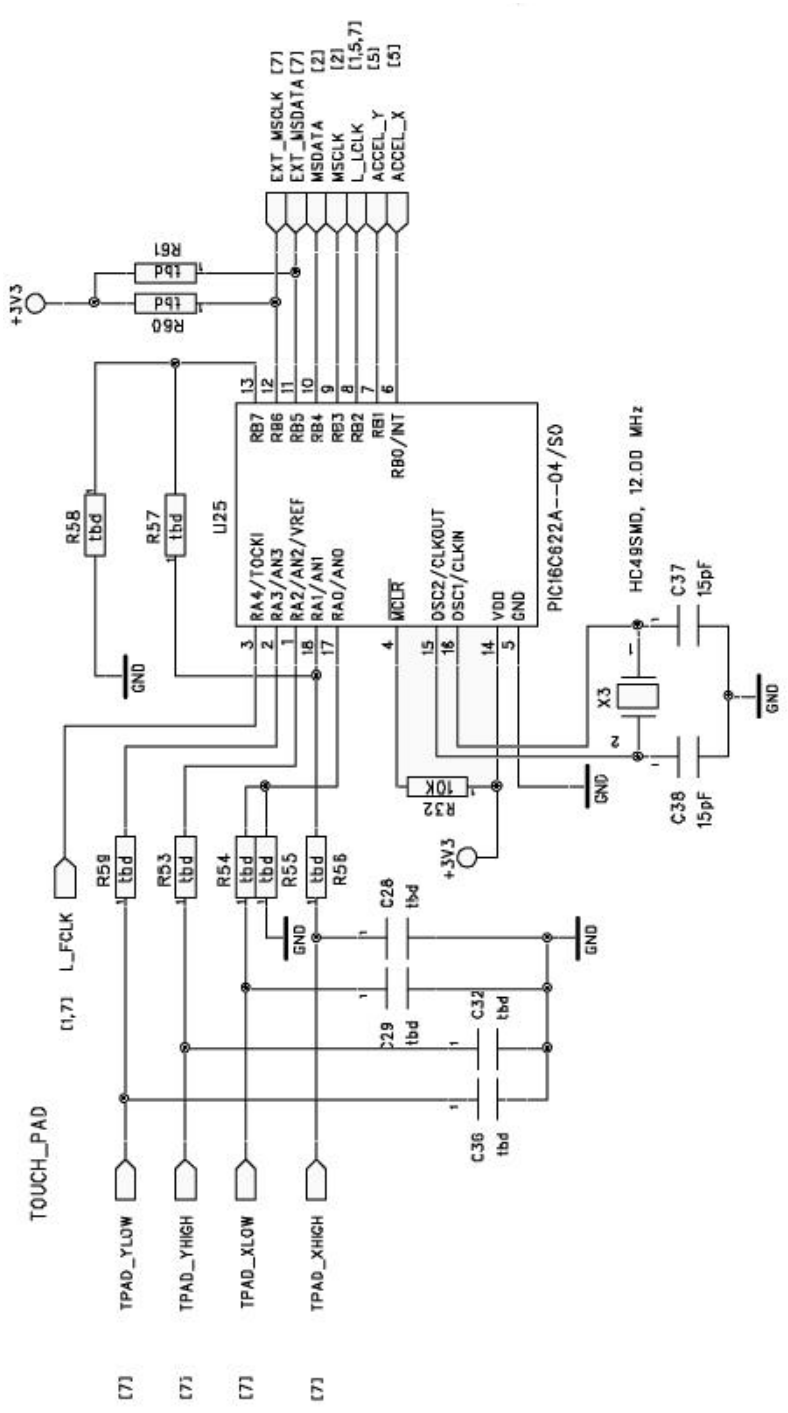

Figure 2.1: The PIC circuit diagram.

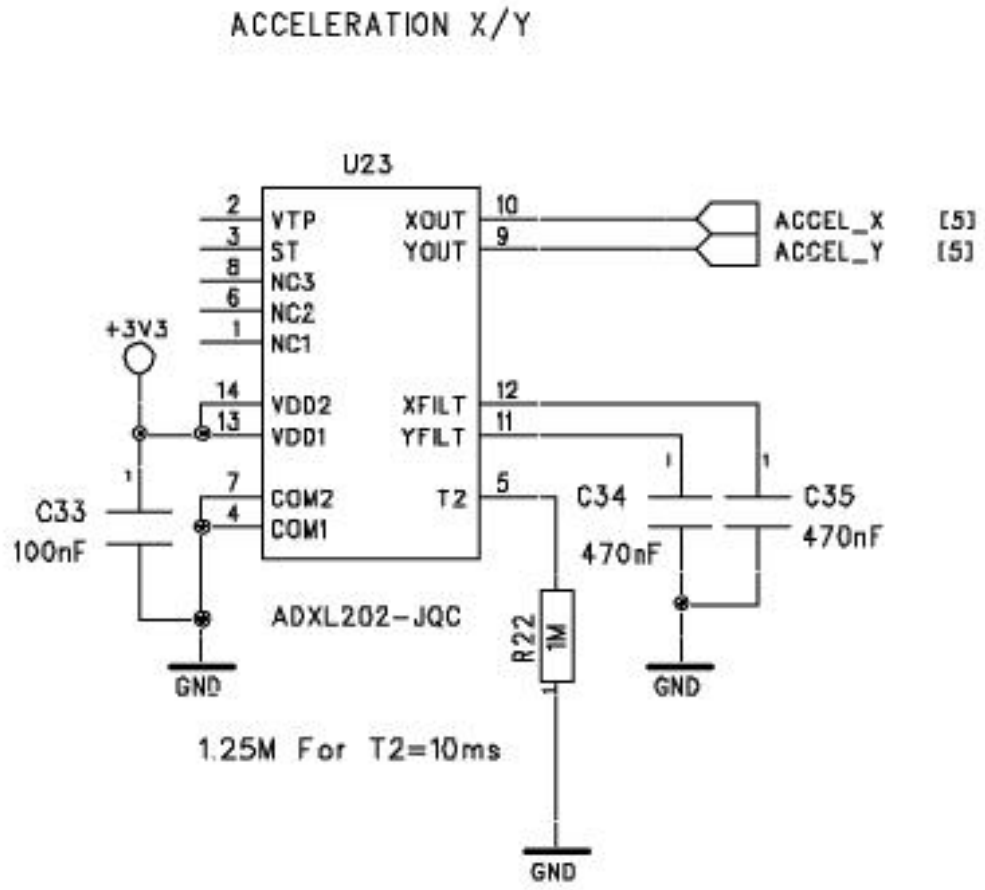

Figure 2.2: The accelerometer circuit diagram. The connection to the PIC is with the connections marked ACCEL_X and ACCEL_Y.

The pinout of the ADXL202 can be seen in Figure 2.3. The pins are summarized in Table 2.1. More information can be found in [15].

The accelerometer can detect accelerations in the range of $\pm 2g$. The acceleration is expressed as a ratio between two times, T_1 and T_2, where T_1 represents a pulse of positive voltage. (See Figure 2.4). When the ratio T_1/T_2 is 0.5, the acceleration is nominally $0g$.

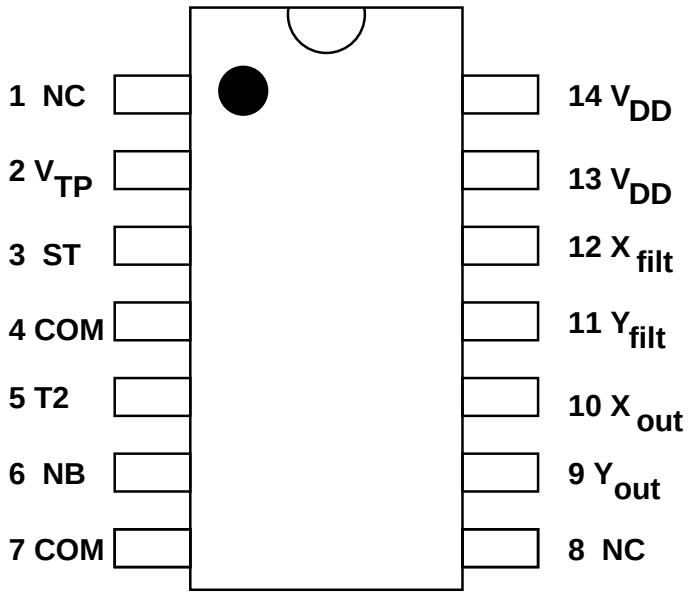

Figure 2.3: ADXL202 pin configuration

As the acceleration range is $\pm 2g$, the span is $4g$. This implies that the acceleration in units of g can be deduced by measuring the length of T_1, the length of T_2, and applying the following formula (found in the ADXL specifications [15]):

$$a = \frac{T_1/T_2 - 0.5}{0.125} = 8\frac{T_1}{T_2} - 4. \tag{2.1}$$

2.3.1 Input pins

The accelerometer gets its power supply through pins 13 and 14. Pin 5 is connected to a resistor, R_{SET} (shown as $R22$ in Figure 2.2), which governs the base duty cycle period T_2. The formula for this is:

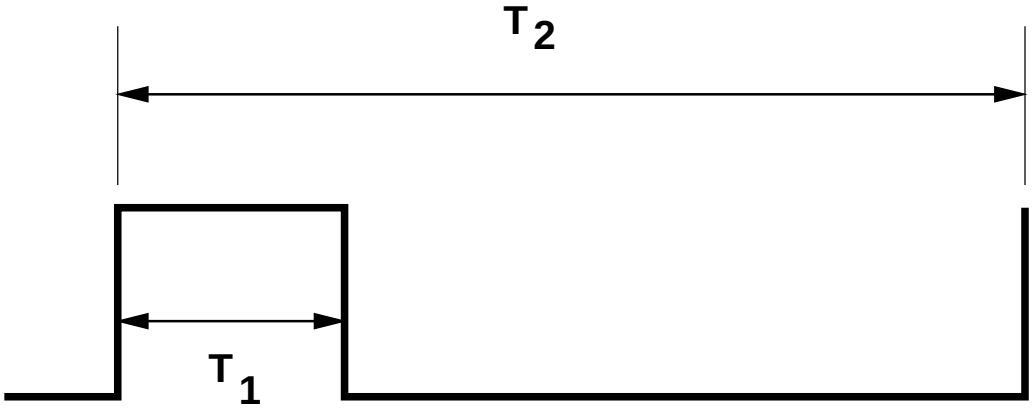

Figure 2.4: Pulse width modulated output – typical duty cycle

Pin	Name	Description
1	NC	No connect
2	V_{TP}	Test point
3	ST	Self test
4	COM	Common
5	T2	Connect R_{SET} to set T2 period
6	NC	No connect
7	COM	Common
8	NC	No connect
9	Y_{OUT}	Y axis duty cycle output
10	X_{OUT}	X axis duty cycle output
11	Y_{FILT}	Connect capacitor for Y filter
12	X_{FILT}	Connect capacitor for X filter
13	V_{DD}	$+3V$ to $+5.25V$, connect to pin 14
14	V_{DD}	$+3V$ to $+5.25V$, connect to pin 13

Table 2.1: ADXL202 pinout

$$T_2 = \frac{R_{SET}(\Omega)}{125M\Omega}. \tag{2.2}$$

A $1M\Omega$ resistor is used, giving a period for the base cycle T_2 of $8ms$, or 125Hz. This means that a null acceleration will result in a pulse length T_1 of $4ms$.

The X_{FILT} and Y_{FILT} pins (numbers 11 and 12) are connected to $470nF$ capacitors. These are used for setting the analog filter bandwidth of the the pins. This affects the resolution capability of the accelerometer, as well as how much noise there is in the signal. The value of 470nF means that the bandwidth will be 10Hz, and the noise will amount to 1.9mg. The peak-to-peak noise value is estimated to be four times the noise value. In this case, it is 7.6mg. These values are from [15], page 8.

Power is supplied to the accelerometer with a $100nF$ capacitor. This evens out the power supply.

2.3.2 Output pins

The output pins (X_{out} and Y_{out}) supply the PWM output to the microcontroller. As the output is digital, i.e. one or zero, it can be input directly to the PIC and its width measured with the PIC's counter routines.

2.4 The PIC

The pinout of the PIC 16C622 is shown in Figure 2.5. Table 2.2 summarizes the connections.

The PIC is used to decode the signals from the accelerometer and the touchpad. By using the PIC, the signals from these devices can be preprocessed before being delivered to the main processor.

The choice of PIC for the accelerometer part of the project was given, as it had already been selected for the touchpad processing.

2.4.1 Input pins

Power is supplied through pin V_{DD}. Pins CLKIN and CLKOUT are connected to a $12MHz$ crystal oscillator. Input from the accelerometer are received through pins RB0 and RB1.

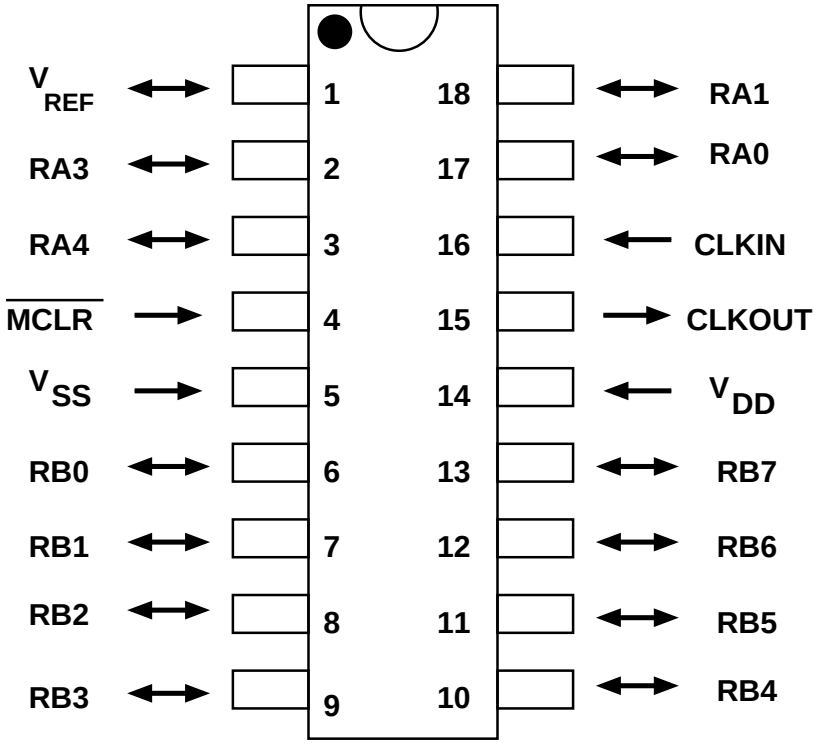

Figure 2.5: PIC pinout, showing the different input/output pairs

Pin	**Name**	**Input** ($\leftarrow$) **output** ($\rightarrow$) **both** ($\leftrightarrow$)
1	RA2 / V_{REF}	$\longleftrightarrow$
2	RA3	$\longleftrightarrow$
3	RA4 / TOCKI	$\longleftrightarrow$
4	$\overline{\text{MCLR}}/V_{PP}$	$\longrightarrow$
5	V_{SS}	$\longrightarrow$
6	RB0 / INT	$\longleftrightarrow$
7	RB1	$\longleftrightarrow$
8	RB2	$\longleftrightarrow$
9	RB3	$\longleftrightarrow$
10	RB4	$\longleftrightarrow$
11	RB5	$\longleftrightarrow$
12	RB6	$\longleftrightarrow$
13	RB7	$\longleftrightarrow$
14	V_{DD}	$\longleftarrow$
15	OSC2 / CLKOUT	$\longrightarrow$
16	OSC / CLKIN	$\longleftarrow$
17	RA0	$\longleftrightarrow$
18	RA1	$\longleftrightarrow$

Table 2.2: PIC 16C622 pinout

2.4.2 Output pins

The PIC communicates with the StrongARM through pins RB3 and RB4. The protocol used is described in Section 3.4.1.

2.5 Test setup

In order to test the micro-controller software and communications with the Linux kernel, a test board was constructed. This permitted hardware debugging and easy reprogramming of the PIC when necessary. Almost all of the development described in this report was made with this equipment.

The hardware was acquired and mounted on a so-called lab board. This is a plastic board with holes permitting easy wiring of small circuits. The wiring followed the diagrams Figure 2.1 and Figure 2.2. Power was supplied by a stable voltage source.

The lab board solution was chosen after a first try using direct soldering of the components to a PCB. This approach made it difficult to correct mistakes, however, and the lab-board solution was adopted instead.

Instead of the production form of the PIC, which is in PCC form, a quartz windowed version (erasable with ultra-violet radiation) was used. This was mounted in a quick release "Zero Insertion Force" (ZIF) socket, permitting easy access. The PIC programming was handled with a device from Microchip, the Pro Mate II programmer [17]. This is a general purpose device which can program most of the PICs from Microchip, in both erasable and one-time programmable configurations. The programmer has different boards for each class of device. It can also program devices which are already surface-mounted. It is integrated with the MPLAB development environment, available at no extra cost from Microchip.

Using the Pro Mate II programmer requires Microsoft Windows. Another simpler programmer from Microchip, the PICSTART Plus [18], can be used with Linux, with the tools available at Andrew Pine's site [19]. More resources for PIC programming with Linux can be found at `http://www.gnupic.org/`.

The ADXL202 is only available in surface-mounted packages. A small PCB was acquired and an accelerometer hand-soldered to it. The PCB was then fitted with pins permitting mounting on the lab board. Care has to be taken when soldering the accelerometer. A number of them were found to be non-operational after mounting. If this was the result of static discharge, excessive acceleration (from the shock of dropping them, for example), or excessive heat when soldering, is not entirely clear. As the only way of

checking the function of each component was by mounting it first, the author was often confronted with hours of finicky work for nothing. It has been brought to his attention that this is more often the case in electronics design than one might think.

The parallel port connector (female DB25) was a standard component. The three wires connecting to it (MSCLOCK, MSDATA, and ground) were soldered on, care taken that they would not fall off if the unit was roughly handled. A standard parallel port cable (a so-called "LapLink" cable) was used between the test board and the PC used as testing machine.

Chapter 3

Software: the micro-controller

The main reason for using a micro-controller between the accelerometer and the host processor and its operating system is that a micro-controller is good at simple things. Using a micro-controller reduces the load on the host processor, leaving more resources for other tasks, which are arguably more important for a user. In this case, a micro-controller takes PWM inputs from the accelerometer, decodes them into acceleration, and send these values to the OS for use in the user interface. This can be done quickly, cheaply, and at low power.

In the Tifón project, there are two micro-controllers on the PCB. One handles the keypad, the other the input from the accelerometer and the touchpad. Both separately communicate with the StrongARM .

The Microchip PIC 16C622 [16] is an 8-bit micro-controller with 2KB of program memory, and 128 bytes of data memory. It has 13 I/O pins, which can be directed individually. It has a 8-bit timer with a programmable prescaler, making it possible to measure times with values larger than can be expressed in 8 bits.

The PIC receives PWM input from the ADXL202. The input consists of a series of pulses, the length of which (denoted T_1) are less than the base period T_2. The ratio T_1/T_2 is proportional to the acceleration sensed by that axis, as shown in Figure 2.4. By measuring T_1 and comparing it to T_2, a value for the acceleration can be deduced. This is then sent to the OS to be used by the terminal.

3.1 Programming language choice

The traditional language of choice when programming a PIC is assembly; it

is the "native" language of the processor [1]. Assembly language has several benefits: it is fast and doesn't have much overhead. However, both the author and Stefan Hellkvist (responsible for the touchpad) were collaborating on one project and one processor, and as both of us were more comfortable working in C, that is the language we chose.

We purchased a C compiler from CCS Inc. [20], which integrated nicely into the integrated development environment (IDE) software provided free of charge by Microchip[2]. This enabled us to work wholly in C. We felt that the advantages of using a higher-level language outweighed the disadvantages of larger code size and greater complexity. These advantage were familiarity, as both developers knew C but not assembly; and functionality: C made it possible to re-use software components between the accelerometer and the touchpad.

The MPLAB IDE is only available for the Microsoft Windows operating system, so instead of using Linux as a development platform, we had to use a PC running Windows 95. This computer also controlled the chip programmer from Microchip, enabling us to program the micro-controllers in the lab.

3.1.1 Limitations in the C compiler

The CCS compiler has a number of limitations compared to standard C. These are a result of the architecture of the PIC. For example, an `int` is 8 bits, whereas in ISO C it is defined to be at least 16 bits. See [21], page 111, for details of the representation of the standard C datatypes. This means that the built-in integer arithmetic isn't enough for the calculations used in the program, and we therefore have to use supplementary functions. The use of these is described in Section 3.3.1.

3.2 Decoding algorithm

The signal from both axes on the accelerometer consists of a square wave, where the pulse length is proportional to the acceleration sensed on that axis. If the base period of the signal is T_2, an acceleration of $0g$ means that the length of the pulse T_1 is $T_2/2$. A maximum positive acceleration means that $T_1 = T_2$, and a maximum negative acceleration is represented by $T_1 = 0$. The ADXL202 can measure accelerations in the range $\pm 2g$. The

[1] Of course, true native code is machine code. Assembly is one step up from machine code.

[2] This software package, MPLAB, is available at `http://www.microchip.com`

duty cycle change per g (where $1g = 9.81m/s^2$) is 12.5%, according to [15]. The acceleration a is then given by Equation 2.1, repeated here for reference:

$$a = 8\frac{T_1}{T_2} - 4. \tag{3.1}$$

A straight-forward way of measuring the acceleration sensed by the two axes x and y is outlined below:

1. Start the timer at the rising edge of the first channel, x.
2. Stop it at the falling edge. This gives T_{1x}.
3. Start the timer at the rising edge of the second channel, y.
4. Stop it at the falling edge; and one gets T_{1y}.
5. Compare these values to $T2$. Repeat.

This is straight-forward, but it has the disadvantage of being slow. One can only get a sample of the acceleration of both channels once every three cycles ($3 \times T_2$). One wastes cycles waiting for the next channel's rising edge, as shown in Figure 3.1

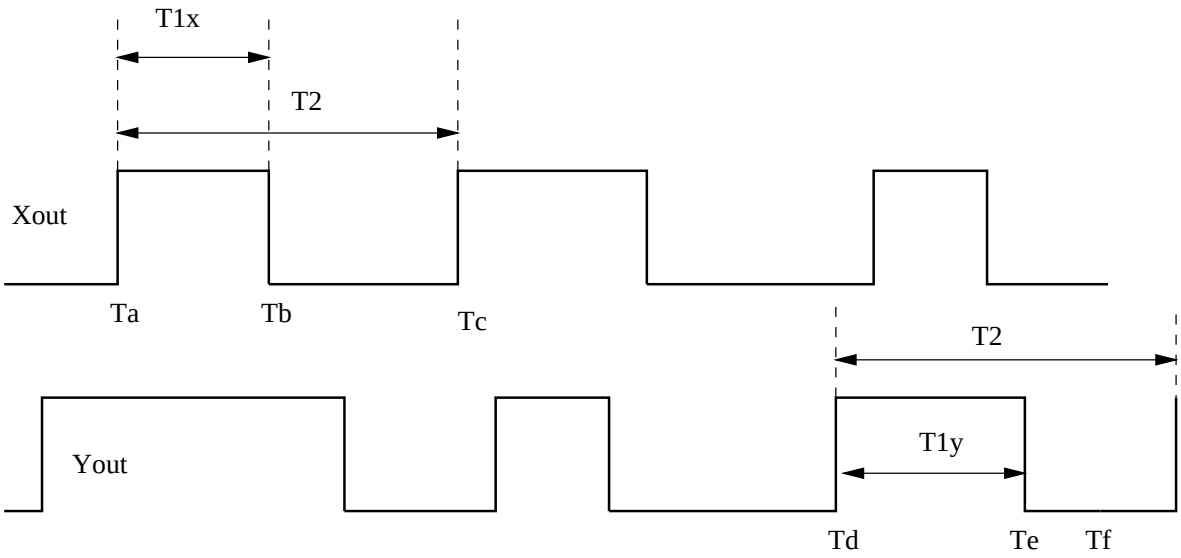

Figure 3.1: Slow decoding scheme.

There is fortunately a solution to this problem. The timing of the pulse output is handled by a triangle wave in the ADXL202. This means that the

midpoint of T_{1x} and $T1_y$ coincide in time. This means that the time between the midpoints of T_{1x} and T_{1y} is equal to T_2. Since we know the value of $T2$, we can use this fact to decode the acceleration in a smaller timeframe than before.

The algorithm is outlined in [22]. In short, we use the following algorithm:

1. Start the timer at the rising edge, T_a of the X channel.
2. Stop the timer at the falling edge, T_b. By definition, T_{1x} is now equal to $T_b - T_a$.
3. We repeat the process for the Y channel, and get $T_1y = T_d - T_c$.
4. As $T_2x = T_2y$, we get $T_e - T_a = T_g - T_f$, and after substitution

$$T2 = T_d - \frac{T_d - T_c}{2} - \frac{T_b}{2}. \tag{3.2}$$

The advantages of this method are twofold: we acquire one sample of acceleration from both axes every two T_2 cycles (compared to three cycles in the previous algorithm); and T_2 is calculated only once for both axes.

3.2.1 Calibration and error correction

The ADXL202 is not a precision device. It is subjected to errors due to drift during operation and variations in temperature. It also does not save it's state between operations. When you start using it, it assumes that it is at rest, even though it may be subjected to acceleration.

Because of this, calibration of the device is necessary before additional operation. The `accel.c` program therefore begins with a calibration routine. Here, reference values are acquired and used in the subsequent calculations to define null acceleration.

The calibration procedure requires the accelerometer to be level and at rest when it is performed. Obviously, in a commercial device, it will be desirable for this to be able to be performed whenever necessary, after a hard reset, for example. However, it is perhaps not optimal to require end users to perform this step. We can therefore foresee a need of a initial calibration at the time of manufacture/assembly, and storage of the pertinent values in non-volatile memory, along with the micro-controller code. This of course adds to the cost and complexity of the device.

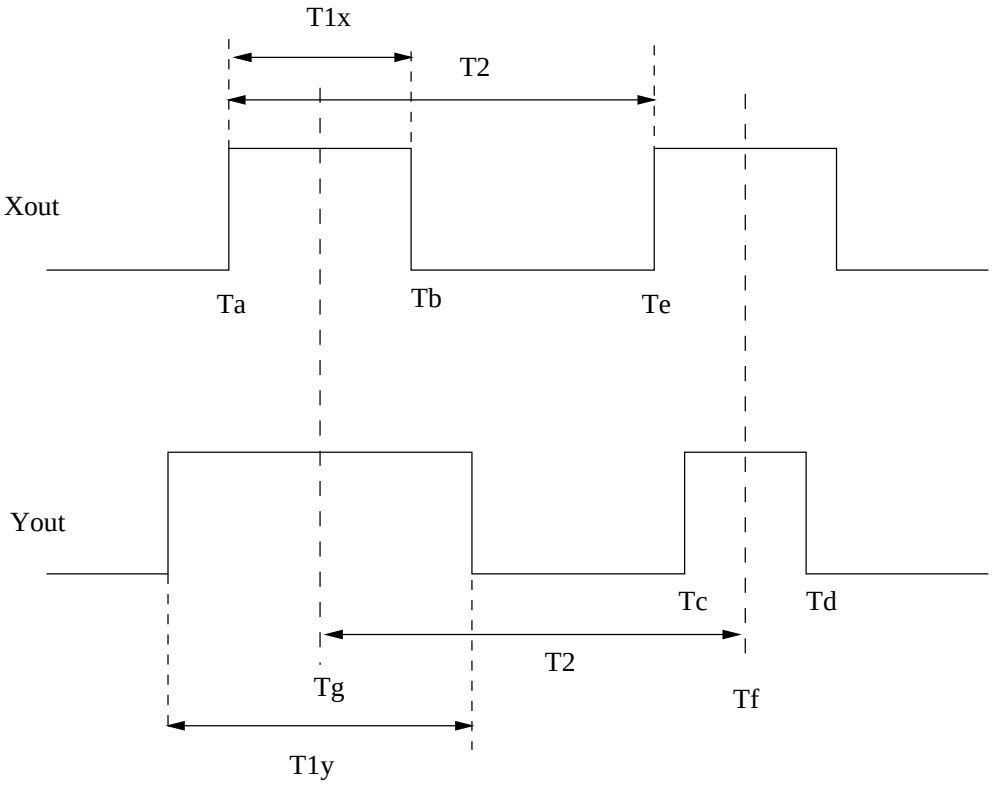

Figure 3.2: Fast decoding scheme.

3.2.2 Correcting for drift and jitter

The values from the calibration are $T_{2\text{cal}}, Z_{x\text{cal}}$ and $Z_{y\text{cal}}$. The value $T_{2\text{cal}}$ is stored as T_2 has temperature drift and jitter. By storing this value and using it in the calculations below, we minimize the effects of these errors.

Two other values are determined during calibration. One is the bit scale factor, or BSF. This is used to determine the bit resolution of the acceleration calculation. Here, the BSF is set equal to 256, which means that an acceleration range of $\pm 1g = \pm 128$ counts. This gives a range of ± 256 for the measurable acceleration, and it fits into an 8-bit number.

The other value is the scale factor, K, used to actually scale the acceleration values into the desired bit range. It is defined as:

$$K = 4\frac{T_{2\text{cal}} \times \text{BSF}}{T_{2\text{cal}}} \tag{3.3}$$

3.3 Acquiring input from the ADXL202

The code referred to in this section can be found in Appendix B.1. Refer to the comments there for additional details.

Now that the ground is laid for the calculation of the acceleration, we need to get raw PWM data from the accelerometer. This is implemented by the function `get_times()` in the main program `accel.c`. First, however, we have to find a method of timing the pulses. This is achieved in the shared code `counter.c`. We use the internal real-time counter (`int_rtcc`) to count processor cycles. As the counter overflows when it reaches 256 (2^8), it triggers an interrupt itself, and we add 256 to the variable `count`. In this way, we can get a value for the counter which is larger than 256.

The function `get_times()` reads the PWM output, times the pulses, and stores the values in the the variables `tb`, `tc`, and `td`. These are used to compute the values of T_{1x}, T_{1y}, and T_2 for this sample. We then correct the measured value of T_2 by calculating a value Z_{actual}:

$$Z_{\text{actual}} = \frac{Z_{\text{cal}} \times T_2}{T_{2\text{cal}}}. \tag{3.4}$$

This is done for both axes.

We then calculate the acceleration from this equation:

$$a = \frac{K(T_1 - Z_{\text{actual}})}{T_{2\text{actual}}}. \tag{3.5}$$

3.3.1 Higher precision math

As previously noted, the C compiler used in the project only has 8-bit integers. This means that multiplications often result in overflows. Therefore, two functions, written in assembler, were used to provide 32-bit multiplication and division. These are in the file `math32.c`. These two functions were originally provided in the CCS compiler distribution, and are reproduced in Appendix B.1.2 with permission.

These routines are needed as there are two multiplications and two divisions for each acceleration calculation of each axis. As the input is no more than 8 bits, and the result likewise is no larger, we do not retain the higher bytes of the intermediate calculations.

Each routine works with a `struct` called `long32` defined thusly:

```
struct long32 {unsigned long lo;  long hi;};
```

A `long` (or `long int`) is 16 bits in this case, so combining two of them into a `struct` gives us a 32-bit container.

3.3.2 Negative values

C uses twos-complement negation of integers. To negate an integer, complement all the bits and add one. For example, -1 is formed by taking the value $(00...0001)_2$, complementing the bits getting $(11...1110)_2$, and adding 1 to the result: $(11...1111)_2$. See [21], page 111.

Negative values of the variable `diff_{x,y}` need to be treated separately. If we have a value of -4 for example, the hex value in a 16-bit variable is `0xfffc` (`0b1111111111111100`). If we don't ensure that this is in fact a negative value, the 32-bit variable in which it will be stored will have the value `0x000000fc` $(256 - 4)$, instead of `0xfffffffc`.

We check for the need to do sign extension by ANDing the value with `0x8000` to see if the highest bit is set. If so, the higher order `long` of the `struct` is set to `0xffff`, otherwise it is zero.

3.4 Data output

As long as the PIC has power, it reads the PWM output from the two axes, compares them to the calibrated values, and computes an 8-bit number representing the acceleration.

We now need to get these values to the next link in the chain from accelerometer to user – in this case, the operating system of the Tifón. We have covered data input and processing. We now turn to data output.

3.4.1 Communication protocol

Many PICs manufactured by Microchip Inc. have a standard serial (RS232) interface built in, which permits use of standard protocols to transfer data at 9600 bits/second – more than enough for our purposes. The StrongARM has a serial interface too, so this would seem to be the ideal way of sending data to the processor.

However, the 16C622 PIC does not have a serial interface. What it does have is two pins which can be controlled by software. These pins are used to implement a home-brewed communications protocol, which is shared by the accelerometer and the touchpad (which is also controlled by the PIC). One of the pins is a timing pin. The other transfers the data.

The reasons for using this protocol instead of a more standard protocol such as SPI or MICROWIRE was that it was used by Stefan Hellkvist for his character recognition system for the touchpad [2].

A custom device driver was written on the Linux side which could interpret the signals and present it to the kernel, which is responsible for handling I/O inside the Tifón. This driver will be presented in the next chapter.

All the data is sent in the form of packets, consisting of 7 bytes. The first byte is the status byte. Its eighth bit is always set, while the data byte's eighth bit is always zero. This means that we only have 7 bits per data byte in which to put data. See Table 3.1.

In order for the kernel to determine which device (accelerometer or touchpad) is sending the packet, the header byte has two device bits (D1 and D0 in Table 3.1). A setting of `0b01` means that the data is from the touchpad; a setting of `0b10` means the accelerometer.

Bits 4 and 3 are used to indicate status (S1, S0) of the touchpad. They are not used by the accelerometer. Neither are bits 0,1, and 2. So if we are sending a status byte for the accelerometer, we set the first byte to `0xc0` (`0b11000000`).

Even though the accelerometer only uses two axes, the touchpad uses three; therefore, we need the bytes for them. When sending packets from the accelerometer, bytes 5 and 6 are reserved for Z-axis data. If, as in this case, there are only two axes, their contents are ignored.

Actually packing the acceleration values into packets is achieved by the function `output_acc()`, which takes two 16-bit `long`s as argument, and relies on the function `writeByte()` to actually toggle the output pins.

This function sends the raw bytes to the processor, along with a parity bit (determined by `isOddParity()`) which is needed by the StrongARM. This processor-dependent feature is replicated by a logic-switch gate in the test setup, as the x86-based PCs used for testing do not need this.

Byte nr.	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	D1	D0	S1	S0	–	–	–
1	0	X6	X5	X4	X3	X2	X1	X0
2	0	X13	X12	X11	X10	X9	X8	X7
3	0	Y6	Y5	Y4	Y3	Y2	Y1	Y0
4	0	Y13	Y12	Y11	Y10	Y9	Y8	Y7
5	0	Z6	Z5	Z4	Z3	Z2	Z1	Z0
6	0	Z13	Z12	Z11	Z10	Z9	Z8	Z7

Table 3.1: Output packet byte ordering

Chapter 4

Software: Linux

After filtering the acceleration values through the PIC, we will use the operating system on the Tifón to provide these values to applications running on the device.

A device driver written for Linux reads the input pins on the StrongARM. This driver splits the input into two streams, one for the touchpad and one for the accelerometer. These streams are presented to the system as standard Unix `/dev/` files, permitting easy access to the data for applications running on the terminal.

4.1 About Linux

Linux is a free, open-source work-alike version of Unix, which was a multi-user, multi-tasking operating system, originally written in the 1970s for mini-computers. Due to its generous licensing and academic popularity, there has been a wealth of software developed for Unix, much of it in the public domain or covered by the Gnu Public License (GPL). This means that the source code is freely available, making it easy to port the software to other architectures.

Linux is a complete rewrite of Unix. It does not incorporate any commercial code, and the kernel is under the GPL. It has benefitted from the extensive testing and refinement of Unix over the years, and works very well as a server OS for small-to-medium sized networks. It can also be used as a workstation; however, a lot of applications written for other commercial variants of Unix are not yet available for it. However, many important tools, such as compilers and debuggers, are free and open-source, and run on Linux.

As was Unix, Linux is based on the C programming language, making programs easy to port to different architectures. Programs can be distributed

in source-code form, and only some site-specific parameters set to compile it on a different platform. C is not a modern language, lacking features such as object orientation, and the structure of the language makes it hard to catch programming mistakes. However, it is widespread, it is easy to do low-level programming in it, and it is fast. This makes it the de-facto standard programming language for basic OS tasks, such as the kernel and networking.

4.2 The case for Linux

Linux is mostly used for servers and workstations running on PC hardware. These are very different from handheld terminals when it comes to power requirements, screen resolution, memory size – and user interaction. Why use Linux in the Tifón, and not an OS specially designed for small devices?

Linux may be mostly used on PCs, but it has been ported to many other platforms, notably the StrongARM. This meant that there was no porting effort necessary for the Tifón. Once the Linux kernel and the Gnu C compiler (gcc) were ported, it was easy to download source code from code repositories on the Internet and compiling it for the Tifón. As a platform for compiling and testing, the Corel Netwinder[1] was used. This is a StrongARMbased "thin client" running Linux. It has standard networking, monitor, mouse, and keyboard ports, so it could be used as a workstation.

The construction of the Linux OS is highly modular. This, coupled with the open-source model, made it easy for the Tifón team to strip unneeded parts from the OS. As an obvious example, a full-blown text editor is not very practical on a handheld terminal. It is not critical to the operation of the device, either, so it can safely be left out.

The fact that Linux/Unix has a desktop-oriented user interface is offset by the possibilities of remote control and the aforementioned modularity. It is relatively simple to implement a different user interface (in the case of the Tifón, this is written in Java) on top of the basic OS. As far as the user is concerned, the fact that the device uses Linux is not even visible. It is used as a stable, open base for other applications running on top of it

Unix is a very good networking OS, as least as far as the TCP/IP protocol suite is concerned. Linux has inherited these strengths. Although there are other networking technologies available which may be better in some ways, TCP/IP is what the Internet runs on. And now, even small applications such as mobile telephones and handheld terminals are expected to interact with the Internet. This makes having a fully working TCP/IP stack in the OS

[1]Since the time of writing, Corel sold its Netwinder product to Rebel.com, which has since been purchased by Zentra. The Netwinder is no longer available.

kernel a good idea. The fact that source code is available means that new protocols[2] are easily accommodated.

All in all, Linux is a better choice for handheld terminal development than might be expected at first. The disadvantage is that the GPL may require any changes made to be offered as source code to the recipient, that is the customer. This can mean that proprietary information developed in-house by Ericsson will be divulged. However, this may mean that the product will be better-supported than others, and therefore be more popular than the competition. Many companies developing products using Linux are actively embracing the open-source philosophy, making certain parts of the OS open to change, while keeping the value-added parts proprietary.

Even if the Tifón is not released under the GPL, the existence of open-source code and tools constitutes a rapid development environment, making it easier and faster to develop in-house products and prototypes. These can then be used as stepping stones on the path to new consumer products.

4.3 Linux on the Tifón

At the time of the project, the hardware on which the OS was to run was not yet completed. This, together with the fact that it is not very convenient to conduct development on what is essentially an embedded system[3], we needed a host machine as development and testing platform.

The version of the Linux kernel running on the Tifón was 2.0.36 at the time of the project. We installed a Intel version of Linux on an IBM ThinkPad 750 as a test platform. This machine was not in active service at Ericsson any longer, so if we made a mistake, not much harm would be done. To communicate with the PIC setup, we used the parallel port with a pass-through cable (a so-called LapLink cable). This was used because it is faster than the serial port, and also because we already had a description of how to use it in [23]. This also illustrates the value of having a sacrificial system. Hooking weird voltages directly into a PC's motherboard can have unfortunate effects. Luckily, we did not have any problems.

Whether using a direct connection to the processor, as in the case of the StrongARM, or by using a parallel cable, as in the case of the IBM

[2]One such protocol is WAP (Wireless Access Protocol) developed by, among others, Ericsson Radio Systems.

[3]As permanent files are kept in non-volatile storage, any change to these files means that they will have to be downloaded from a host machine, and a reboot executed. As the total code was up to 6 megabytes and the only way of transferring it to the Tifón was by a serial link of 9600 bps, this took considerable time.

laptop, the eventual fate of the packets from the PIC is the same. Based on the information in the header byte of each packet, they are divided into packets from the touchpad and packets from the accelerometer. These are then directed to a named pipe, and presented to the OS as a so-called device file (in Unix, everything is a file); in the case of the accelerometer, called `/dev/accel`. This file is a data source – other programs can read the packets coming from it, and use them as they see fit.

The data coming from `/dev/accel` is raw and unfiltered – it has a header byte, and six payload bytes. These have, in the case of the accelerometer, two 8-bit numbers, with values between 0 and 255. In order to present these to the software running on the Tifón, they have to be transformed into more usable values.

At the time of the project, no software existed on the Tifón, period. In order to test the output from the test setup, a simple set of filters were written in order to study the accelerometer. These are described in Section 4.6.

4.4 Parallel cable connection

Details on how to configure Linux for use of the parallel port can be found in [24]. The LapLink cable has some cross-coupled wires. In Table 4.1, we see that the pins 4 and 5 on the device side are connected to pins 12 and 10, respectively.

For the connection between the test board and the Linux laptop, we needed three wires: one for the timing line, one for the data line, and one for ground. The connections used were:

Line	**Pin on test device**	**Pin on parallel port**
GND	25	25
MSCLK	5	10 (ACK)
MSDATA	4	12

Table 4.1: Parallel port connections

The MSCLK pin is important, as the kernel is configured in such a way that it generates an interrupt when the electrical signal on it changes from low to high ([23], page 179).

4.5 Structure of the Linux device driver

The C code referenced in the following can be found in Appendix B.3.

The device driver is in the form of a dynamically loadable module. The `struct pic_fops` defines the file operations of the device. These are read, write, open, and release. 'Write' is included for completeness, but it is not really possible to write to the device.

Each file operation has an associated function. The functions `pic_open()` and `pic_release()` simply increment and decrement the device use count, respectively. However, during testing, this feature caused errors which made it impossible to release the device. Thus, the calls to the functions `MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT` are commented out.

The function `pic_read()` reads from the character array `pic_buffer`. If this buffer does not contain any data, it waits until it does. This is to ensure that proper blocking takes place even when there is nothing to read. Otherwise, a read request can hang indefinitely. If there is data in the buffer, it is copied to the device file `/dev/pic`.

This device is created with a call to the function `register_chrdev()`. It has a major number of 100, defined in `PIC_MAJOR`. This major number was chosen arbitrarily, but according the the *Documentation/devices.txt* in the Linux kernel source tree, it is used by the Linux Telephony devices. Techniques exist for choosing a major number dynamically, see Rubini [23] for details.

This means that when the PIC is sending data, it comes out at `/dev/pic`. Any program (for example **cat(1)**, which simply reads a stream of bytes from a file) can access this file. This means that a clean implementation is easy to set up, in the context of the common Unix metaphor of pipes[4].

The function `intrpt_routine()` is the heart of the module. It is invoked whenever an interrupt is generated. This is done whenever there is a positive edge on the MSCLK pin, as defined in the function call `request_irq`. When this happens, `intrpt_routine()` waits until a positive signal is received from MSDATA. When it is, the data bits are stored in the character `cur_byte`. The parity and stop bits are read as well, although they are not used when the architecture is x86. They are required for the StrongARM. However, as the code was to run on a StrongARM machine, it makes sense to incorporate this feature into the code.

[4]In Unix, it is common to "pipe" (using the pipe character `|`) output from a source through another program which acts as a filter.

4.6 Presenting the data

In order to verify that the data coming from the ADXL202 was correct, a simple program called `acc_display.c` was written. This basically reads from the device file `/dev/pic`, converts the values from the $[0, 255]$ range to $[-127, 127]$ range, and formats the output. The non-existent z-axis is also shown. This gives a form of error checking. If these values are non-zero, something is wrong.

After compiling, `acc_display` is used in the following manner:

```
$ cat /dev/pic | acc_display ,
```

assuming the program is in the $PATH. This command filters the output from `/dev/pic` through `acc_display`.

A typical display looks like

```
x: 78
y: 103
z: 0
```

There should be no major problems in converting this display to a graphical display with the help of, for example, Tcl/Tk, but this was not implemented.

Chapter 5

Results

This chapter attempts to summarize the results of this project. In short, it was successful. We have a working prototype and code, ready for implementation in the Tifón hardware. In this, the author has fulfilled the goals of Ericsson Radio Systems.

However, we have only shown it to work in the lab. The code is not implemented in the actual Tifón hardware. We do not have a user interface to test the use of the accelerometer as an interface tool. Thus, we do not know what kind of enhancements have to be made for the accelerometer to come to its full potential, or even some limited use.

With this in mind, we will detail the results produced so far. An outline of what might have been done is given in the next chapter.

5.0.1 Resolution

The bit scale factor is chosen as 256, which means that the possible acceleration range of $[-2, 2]g$ is represented by a value in the range $[0, 255]$. Filtered through `acc_display`, the range is shifted to $[-127, 127]$.

Tilting the test board showed that it accurately measured the effects of gravity. When tilted upright, the value was either -64 or 64. We thus have a value of $\frac{1}{64} = 0.0156$ g/count.

5.0.2 Jitter

The values at rest were subject to jitter. The value at rest typically varied by 0 to 7 counts rapidly. It only affected one channel, alternating from time to time. It was not clear on what it depended on, or if it was an artifact of that particular accelerometer. However, it does seem as if some kind of

filter will have to be incorporated into the final product which screens out this effect, possibly by averaging several readings over time.

It is not known if this jitter would affect the planned tilt-to-scroll functionality of the Tifón. Also not known is the effects of thermal conditions, specifically draughts. Temperature can be measured with the help of the duty cycle output of the ADXL202, (Weinberg [25]). This information can be used to correct for thermal effects on the accelerometer.

5.1 Implementation details

One thing that surprised the author was how sensitive the device was to the effects of gravity. It should not have been a surprise, as the device functioned exactly as specified, but it was still a revelation to hold the test board, tilt it, and see the values slide all over the place. The immediate conclusion was that the original thought of using the accelerometer-equipped Tifón as a mouse, moving it in a plane perpendicular to gravity, was not practical.[1]

It would have required a plane surface on which to rest the terminal, clearly not a good requirement for a handheld user interface. The primary way of interacting with the terminal through the accelerometer would be by tilting. Here too a problem would manifest itself. A default viewing angle would have to be defined when the accelerometer was accessed, which would then be the plane from which deviations would be measured.

A user would hold the terminal in a comfortable position, and press a button to use the accelerometer feature. The software would recognize that this is a starting state, and therefore define the value pair from the accelerometer to represent zero acceleration. When the user tilts the Tifón, the cursor would move with it, guided by the tilt, much like a popular children's puzzle where the goal is to guide a ball through a labyrinth by tilting a plane.[2]

[1] The use of a three-axis accelerometer would alleviate this problem. Three axes would enable the software to detect the presence of the constant gravitational field, and to use this known quantity as a reference for determining the position and movement of the device.

[2] For an implementation of this game on the Palm, with optional accelerometer support, see Harbaum [26].

Chapter 6

Future work

When this project was first discussed, a more ambitious result was planned. This was to use the code and concepts developed for the Tifón project to build and evaluate a device, using accelerometers, for three-dimensional motion detection. This could be used as a 3D-mouse, for example, or as a game controller, or as an aid in virtual reality environments. It could also be used as an enhanced user interface tool for future handheld devices.

Unfortunately, time constraints hindered me from concluding this part of the project in full. As a matter of completeness, I would like to present what I was able to conclude, as a pointer and inspiration for future work.

6.1 Three-dimensional inertial sensing: an overview

The device used in the Tifón project can sense acceleration in only two dimensions. This means that it is effective in a plane only. However, there is one given value of acceleration in Earth-bound applications: the gravity field. It is possible to infer the device's orientation with respect to gravity, thus enabling tilt sensing, for example.

However, if one wants to completely model the orientation and movement of a three-dimensional body, one needs to measure more than two dimensions. Adding a third axis to the first will give a third dimension, but even if this third axis does not pass through the origin of the first two, the three sensors combined will not give enough information on rotation to give an accurate picture of the device's movement.

In mechanics, the concept of degrees of freedom is central. A point body on a one-dimensional line has one degree of freedom, backwards and forwards. If it is constrained to a plane, it gets another degree, left and right. However,

if it is itself two-dimensional, and we wish to model the rotation around it's axis, another degree of freedom is required. A point body in three-dimensional space has three degrees of freedom too; but if it has dimensions, we need three more degrees to represent the rotation around three axes. Thus, to completely model a body in three dimensions, six degrees of freedom are needed.

Ideally, one only needs one device: this should incorporate three linear accelerometers, and three rotational accelerometers. However, such a device, if it exists, is bulky, and likely to be expensive. [1] A better solution is to combine a number of conventional accelerometers. As we had used the ADXL202 before and had tools for acquiring acceleration values, it seemed a good choice.

But the ADXL202 doesn't have a rotational accelerometer. How can we combine two or more such devices and still get the information we need?

The answer is to place them at a (known) distance from each other. In this case, a rotation around an axis will translate into acceleration along one of the axes of one or more accelerometers. By measuring the acceleration sensed by each accelerometer and collating these measurements, a pure rotation can be deduced from two linear accelerations.

In light of the foregoing discussion, it should be clear that we need three ADXL202 accelerometers. The configuration sketched by the author can be seen in Figure 6.1. In this figure, the gray lines represent a framework, as rigid as possible. The black arrows represent the accelerometers, each with two sensing axes[2].

A schematic of the prototype is shown in Figure 6.1.

We see that the axes pointing into the plane each define a unique vector. Thus, they span a three-dimensional space. The ones pointing inwards, along the axes, also span space. Thus, we have two redundant sets of vectors (redundantly spanning space, that is.).

The inward-pointing axes are used primarily to provide information on translational movements, while the ones pointing at right angles are used for rotational information.

Each translational axis has a rotational one pointing in the opposite direction. Redundancy is good here, as it can provide sanity checking for some data.

[1] Since the writing of these lines, a self-contained sensor package with the stated characteristics has been built by Ari Benbasat. It is described in [14].

[2] It can be argued that it would have been cleaner to have the accelerometer axes pointing along the major axes pointing *outward* instead of inward. However, this is the configuration used in the prototype which was built by the author. Mostly the directions were chosen so as to make the wiring simpler during construction.

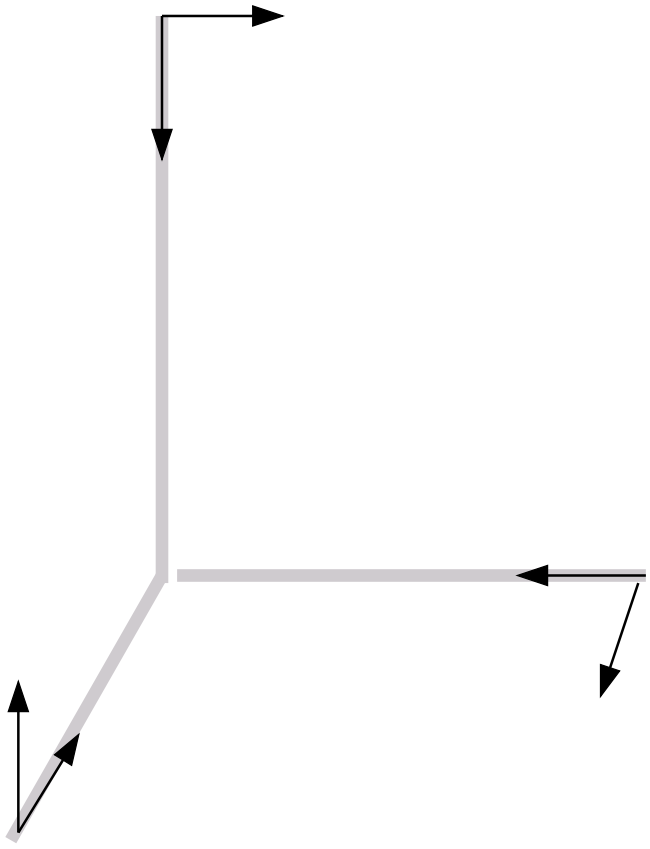

Figure 6.1: Schematic sketch of a multi-dimensional position-sensing device.

Each accelerometer is coupled to a PIC, running a modified version of the software described in Chapter 3[3]. Essentially, it reads the values of each accelerometer, stores them, and sends them to the controlling computer (running Linux) in one big packet. The heavy processing, that is the actual configuration calculations, is carried out there.

6.2 Issues in inertial navigation

If we had no gravity, inertial navigation (at least) would be much simpler. However, on Earth, there is always an acceleration of about $1g$ present. In a system designed to navigate on a plane surface, this is no problem. But if the goal is to arbitrarily track movement and orientation in space, it can be confusing. How do we separate the gravitational acceleration from the translational/rotational acceleration?

We have one fact on our side: gravity is (locally) constant. If we can ensure that we at all times know it's value and can deduct it's contribution from our calculations, we are done.

One way of achieving this is outlined in the following algorithm sketch:

1. Start in a known state: known velocity and orientation.
2. Calculate the direction of gravity. If we start from rest or a constant velocity, this is the only acceleration present.
3. Start the position calculation. Register the acceleration, integrate the position/orientation from this information.
4. Deduct the contribution of gravity.
5. Use the new position/orientation as the new known state; repeat.

The significant part of this scheme is step 3. Obviously, this is not a trivial calculation. We have acceleration information from six sources, but they only define three directions. Three of these sources have to be incorporated into a rotational framework, and the contribution to rotation calculated from that.

In this, we are aided by the things we know. We know (or can assume with sufficient precision) that the framework to which the accelerometers are fixed is rigid. We know the exact position and angle of each accelerometer with respect to this framework. With the help of this information, we can collate the acceleration measurements and eliminate certain configurations:

[3]This program is listed in Appendix B.2

for example, that each accelerometer is receding from the origin at a constant speed.

With the help of this information we can construct a model of the current position and orientation. The accuracy is limited by the accuracy of the accelerometers. Unfortunately, these are not very accurate. If the acceleration error is Δa, integration with respect to time will yield a position error Δx:

$$\Delta x = \Delta a(\frac{t^2}{2} + t) \tag{6.1}$$

At the specified Δa of $7.6 \times 10^{-3}g$, or $0.075m/s^2$, Δx is $1.3m$ after 5 seconds, and $139m$ after 1 minute.

6.3 Another approach

An approach which integrates the sensor and controlling computer in one package is the Compaq iPAQ [30], equipped with a PC Card expansion pack, and a PC card (PCMCIA card) containing the accelerometer and controller.

The limited size of the card would mean that only a 3-axis accelerometer could be used, but that is enough to research many of the areas outlined in the next chapter.

The choice of the iPAQ means that one can use either the native Windows CE OS, or the Linux distribution available at `http://www.handhelds.org`.

Using the PC card interface eliminates wires, and provides the desired rigidity between sensors and the rest of the device. Using the RS323 serial protocol makes it easy to adapt existing drivers to the new task. Depending on the size of the accelerometer, part of the circuitry could protrude outside the card itself, in analogy with the antenna of certain PC cards for wireless communication.

Applications that take advantage of the accelerometer data can be developed separately in a cross-compiler environment on a workstation.

For a solution using a 2-axis accelerometer communicating with the serial port of an iPAQ running Linux, see Van Laerhoven's notes at [27].

Chapter 7

Possible applications and summary

In the preceding chapters, I have outlined a possible implementation of an accelerometer in a handheld device. As of the writing of this document, no commercial devices have emerged using accelerometers for gestural input. I would like to devote this chapter to speculation on how accelerometers could be used to enhance handheld devices.

I will present a number of use cases, i.e. a short description of a possible use of a handheld device with accelerometers and related software.

7.1 PDA with attitude sensing

Personal digital assistants (PDAs), such as the Palm Pilot from Palm [28] and the various devices using Microsoft's Windows CE operating system [29], are small handheld computers typically used for storing contact information and calendars.

As their computing power increases, more and more functionality can be added, including a colour screen, multimedia features, and sound. The Compaq iPAQ [30] is a good example of this type of PDA.

Generally, these devices have a large display, typically taller than wider. Below the screen are a number of buttons linked to common functions and perhaps a pointing device for on-screen navigation. For an illustration of a hypothetical device, see Figure 7.1.

7.1.1 Reading text

High-resolution colour screens make reading electronic text easier – colour

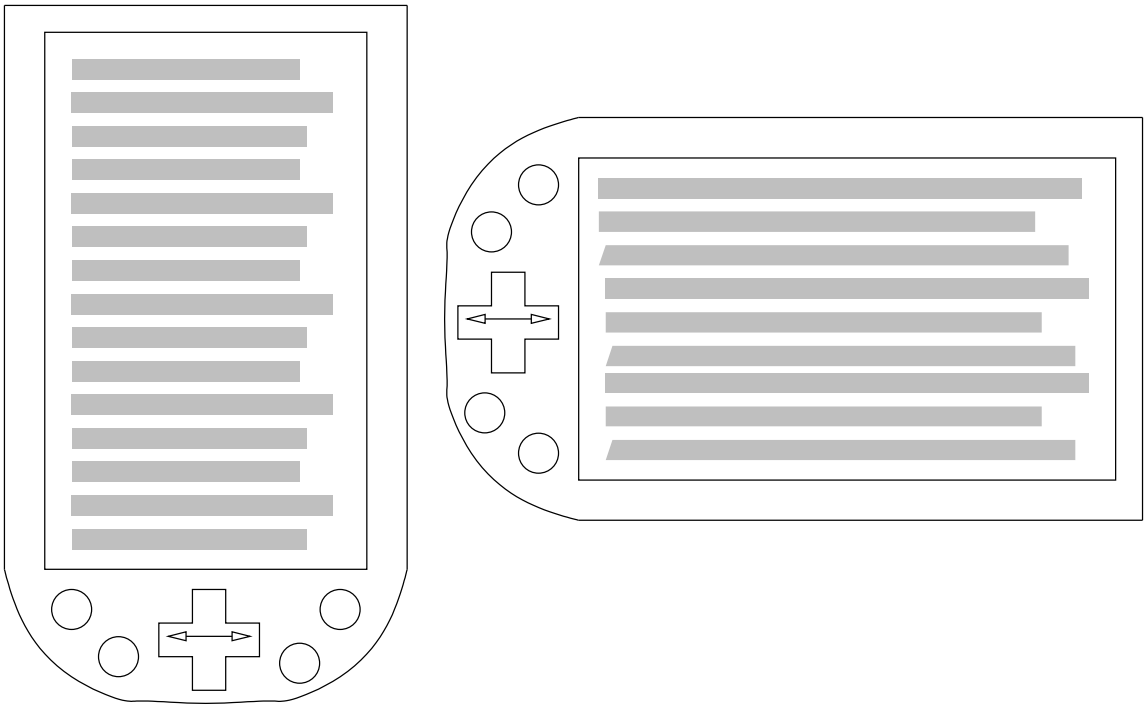

Figure 7.1: A hypothetical PDA, shown in the default upright orientation (right), and tilted for a wider screen (left). The arrows shows the arms of the cross-pad used for scrolling text.

(or more properly) can be utilized for anti-aliasing, rendering on-screen fonts smoother to the eye. A widely spread application for reading electronic text is Microsoft's Reader, [1] available for the iPAQ.

Turning the device on its side will present a wider aspect ratio, enabling more text per line, or perhaps a larger font. An accelerometer can be used to measure the direction of the gravitational acceleration, and reconfigure the screen accordingly. A device with this functionality is described in Hinckley et al. [11].

The user only need turn the device 90 degrees, and the display will automatically adjust itself. The buttons and navigation aid can still be used. Of course, their functions can also be reconfigured, so that the action of scrolling through text is "natural". For example, one e-text reader might use the up-down function of the navigation device to scroll through text. When the device is rotated, the left-right function could be used instead.

A forward looking manufacturer will ensure that tilting either to the left or right is possible, to accommodate left-handed users. Even upside-down use can be implemented, with the screen inverted: this could be used for showing the contents of the screen to someone else.

The above behaviour can be desirable in other situations, such as viewing large calendars, playing games, viewing photos, or watching movies.

7.1.2 The use of accelerometers

The functionality described above can be achieved with the help of an accelerometer such as the ADXL202. In this case, two sensing axes are enough, as the only acceleration considered is gravity.

The software must be able to discard accelerations due to sideways motion. It should also be considerate, and not change screen orientation too quickly. These two requirements are simultaneously achieved by delaying the reorientation by a certain amount of time, and averaging the sensor readings over a longer time interval.

User testing is required to ensure that this delay is perceived as correct. On one hand, it should not be too short, as the risk of incorrect reorientation increases. On the other, it should not be too long, as a first-time user could miss this feature while handling the device. In Hinckley, the device has to be in a specific orientation for 0.5 seconds before changing aspect.

[1] `http://www.microsoft.com/reader/default.asp`

7.1.3 Calibration

The Palm Pilot introduced the concept of a docking station, used for recharging the devices batteries and for data exchange. When the device is docked, its orientation is known (i.e., it is typically upright). The accelerometer can be calibrated by measuring gravity and aligning it with the accelerometer's orientation within the device.

7.1.4 Power and processing requirements

Generally, a iPAQ-class PDA has plenty of computing power (the latest model, the 3800, has a 206 MHz StrongARM processor with 64 MB of RAM). There should be no problems dealing with the data from the accelerometer.

Power drain for this subsystem should be negligible in comparison with the power drain of the display and audio subsystem. Wireless networking, if implemented, will also drain power. However, the accelerometer may have to be active even when the device is inactive ("standby"), thus reducing standby time. However, power consumption can be lowered by letting the accelerometer sleep after a longer period of immobility, waking on movement.

7.1.5 Conclusion

The use of an accelerometer in this case is a good example of making a device more helpful. The action is non-intrusive and intuitive. The user does not have to master complex commands to reorient the screen when needed – the device does this automatically.

7.2 Mobile phone with motion sensing

Mobile phones are generally used in a different way than PDAs. They are always on, have a smaller form factor[2], and are predominantly used for making and receiving phone calls.

These factors mean that a manufacturer has less room physically to incorporate an accelerometer into a mobile phone, compared to a PDA. However, the benefits of using input methods other than the often minimal keyboard are much greater, as they can make the device easier to use.

In the following, three different cases will be described, along with the conditions necessary for realizing them.

[2]Modern phones seem to have converged to the dimensions of a Nokia 8210 (`http://www.nokia.com/phones/8210/`), around $10\text{cm} \times 4\text{cm} \times 2\text{cm} = 80\text{cm}^3$.

7.2.1 Declining a call

If the phone is fitted with an accelerometer capable of measuring the direction of gravity, the orientation of the phone can be determined. One way of exploiting this is by letting the user decline a call by turning the phone over, if it is lying on a table.

A two-axis accelerometer is sufficient in this limited case. The resolution needed is not great; essentially, only the direction of gravity needs to be detected. Calibration can be performed by having the user access a menu item while the phone is lying in a defined state (right-side up on a level surface).

Although simple to implement, this use case is not very flexible. Some people never put their phones on the table. Some users want to see who's calling before declining a call, general good manners require all phones to be switched off during meetings, etc. However, this could be part of the repertoire of a more capable accelerometer-equipped phone.

7.2.2 Unlocking a phone

The author's current mobile, a Siemens SL45 [3], has the keypad lock button awkwardly placed near the lower edge of the phone. It is not easy to lock and unlock the phone, something that is done quite often during the use of the phone.

This led to the thought of using a gesture similar to the unlocking of a key to unlock the keypad. The user holds the phone in the palm of his or her hand and turns is counter-clockwise to imitate the use of a key.

For this to work, a gyroscopic sensor is needed to recognize the turn around an axis[4]. The resolution of a typical accelerometer is not high enough to detect a twisting motion, even if the accelerometer is situated as far from the turning axis as possible.

The same functionality can be achieved with a phone equipped with a three-axis accelerometer, if the gesture is performed as rotating the phone 360 degrees along its long axis.

Is this a good application for a gestural interface? It depends on how easy it is to recognize a deliberate unlock from the usual motion of a phone in use. If the phone unlocks the keypad because the user has turned around suddenly, the user is not going to appreciate the feature. On the other hand,

[3] `http://www.siemens.com/sl45`

[4] Gyroscopes are not covered in this article, but the interesting work by Benbasat [14] covers the use of the Murata ENC-03J model in a six-degree of freedom sensor.

if the motion needed to unlock the keys is too strenuous, the feature will not be used either.

A good candidate for a gesture needs to be easy to remember, easy to perform, and unlikely to be made by mistake. In the above case, a requirement that the phone should be held more or less horizontally could eliminate the false positive gesture made by the user turning around with the phone held vertically in a pocket.

A simpler way to unlock the keypad could be via a voice command, or simply moving the unlock key to a better place on the phone.

7.2.3 Taking a call

A third application, originally suggested by Gerald Maguire, is to use a three-axis accelerometer to determine the path taken by a phone when a user lifts it to his or her ear. This can be used to automatically take a call, without pressing any buttons.

Part of the attraction of this gesture is that it mimics the action of answering a desktop phone. The action is already familiar to the user. Another advantage is that it saves the user time and a key-press to do something that is probably done several times a day.

This type of functionality minimally needs a three-axis accelerometer to be implemented. The problem thus devolves to tracing a point in three dimensions, coupled with the orientation of the device with respect to gravity. Although more degrees of motion can be measured with gyroscopic sensors, the author feels that the added complexity, size requirements, and power drain mean that a six-degree of freedom measurement system is too unwieldy to be used in a mobile phone.

Hinckley et al. [11] describe a similar functionality, that of automatically turning on the voice recording facility of a PDA when it was held at a certain angle near the user's mouth. This was achieved with a 2-axis accelerometer in conjunction with an infrared proximity sensor.

The Hinckley device relied on the fact that the user is prepared to wait for the device to recognize the combination of tilt angle and proximity, along with a time-out of 0.5 seconds to minimize false positives. This is not a good method for taking a call, as holding a ringing phone close to the ear is irritating. A phone capable of recognizing a call-taking gesture thus needs to speculatively recognize the desired gesture, to minimize the delay between gesture and action.

Accuracy and calibration

The acceleration measuring system in the phone must be able to distinguish the motion caused by the user lifting the phone to a listening position from other motions. As the functionality only needs to be present when the phone is ringing, the solution space that has to be searched is limited in time (measurement only needs to be done when the phone is ringing). Similarly, power can be conserved by only using the accelerometer during the ringing phase.

Calibration can be done with respect to gravity, either with the phone lying prone or in a charging dock. If significant differences of performance are expected for each axis of the measurement system, a more elaborate calibration routine may be required. This can be alleviated with special stand or cradle, supplied with the phone.

Recording the gesture itself will also have to be done in a controlled manner. Perhaps doing it twice, in analogy with passwords, will be enough to supply the phone with enough data to recognize the gesture.

Further applications

If the above technology is incorporated into a phone, it can also be used to record gestures for various actions, such as calling a number or locking/unlocking a phone.

For example, a user enters movement record mode, and moves the phone in a clockwise circle. After recording the gesture, he can choose to associate it with calling the office. After that, simply performing the gesture will place the call.

Power and processing requirements

Assessing the amount of computer and electrical power needed by a phone equipped with accelerometers is not easy. The author thinks that the processors of mobile phone will need to be faster than they are today – perhaps as fast as a StrongARM. This means that the power requirements will be larger, implying batteries or other power supplies that are more powerful than today's, yet with the same general dimensions and weight.

It is therefore hard to predict just how much power a phone with a gestural interface will devote to that specific subsystem, as advances in processor technology will also affect the design of MEMS sensors.

An alternative to a built-in system could be supplementary cards, such as a 3-DOF version of the MotionSense SD (Secure Digital) card for the Palm

PDA system.[5]. Such a system would give manufacturers, developers, and users some choice in which system to implement, and how to balance the equation of computing power, energy usage, and added features.

7.2.4 Discussion

Using accelerometers can make a device "smarter", by providing it the means of recognizing how it is held, or how it is being moved about. By integrating this knowledge into intelligent, non-intrusive user interfaces, the manufacturers of mobile devices can enhance their products and make them easier to use.

The minimum requirements for a versatile gestural interface is a system of accelerometers that can track motion in three dimensions. Using gyroscopes to measure rotation expands the amount of gestures that can be recognized, but the larger size of today's gyroscopic sensors means that the devices that have the largest advantage of gestural recognition – mobile phones – also have the least space for accommodating them.

In summary, applying motion sensing and gesture recognition in a mobile phone would require the following characteristics.

- Minimally, three degree-of-freedom sensing. Using accelerometers to measure the movement of a point in space is generally more useful than measuring rotation around a point with gyroscopes. A full six degree-of-freedom system would be capable of recognizing even more gestures, but would be too bulky to implement with present technology.

- A fast, energy-efficient method of recognizing gestures. Fast because response time is important when using a mobile phone, and a gestural interface should be easier and less intrusive than alternative interaction methods. Energy-efficient because battery life is a selling point with mobiles, and a general gestural interface would need to be running all the time, separating gestures from the other motions of the phone.

- A simple method of user customization. The phone could be delivered with some preset gestures, such as tilting for panning, and perhaps an lock/unlock function, but it should be easy for the user to record their own gestures, thereby personalizing the device for their own use.

An interesting discussion in the future of user interfaces, especially with respect to application developers, can be found in Myers [3].

[5]MotionSense FPSA Controller, `http://www.motionsense.com/product/index.html`. This device is designed to be used with map-viewing software for the Palm.

7.3 Mobile phone with position sensing

One application for inertial sensing in mobile devices is gestural input. Another application is using inertial sensing for determining position.

Using inertial sensors and starting from a known position, software in the handheld device can continuously determine its position through integration. By combining this information with internal maps or GPS navigation, the device can be programmed to perform tasks depending on it's location.

7.3.1 Quiet theatres

One application for this could be enforcing mobile-free zones in theatres and restaurants. A phone equipped with an inertial navigation system would know when it was in a quiet zone and disable the ring tone or automatically forward all calls to an answering service. Highly sensitive areas, such as hospitals, could switch off offending phones capacity to radiate radio signals, by detecting when a phone is coming close to sensitive medical devices.

Other methods of achieving this would be by broadcasting a signal that the phones would acknowledge. The advantages of an inertial navigation system would be that the control of the phones behaviour could be more fine-grained. There would be no bleed-over of the signal to areas, such as lobbies, where phone use would be accepted.

Further, it would be much easier for small establishments, such as cafeś, to add themselves to the system. There would be no need for expensive infrastructure investments for each location.

Calibration

A disadvantage over a broadcast-based control system is that each device must be know its position accurately. However, due to the fact that errors in the accelerometer output are integrated along with the true signal, the error in position is large after a short time, in the order of more than 10 seconds. See Section 6.2 for more details.

Some way of recalibrating the output from the accelerometers must be present. Using the Earth's gravity field is one way. In the presence of constant acceleration, the device is assumed to be at rest, and the output normalized.

External sources, such as the Global Positioning System (GPS) can be used to determine the position of the device, and recalibrate the accelerometer's output.[6] Another method is calibrating position with the help of radio

[6] A system for using magnetometers to augment the capabilities of GPS receivers in case of signal loss is briefly described at `http://www.ssec.honeywell.com/quick.html`.

or infrared beacons. For an interesting insight into the infrastructure involved in such a system, see the Websign project at Hewlett-Packard (Pradhan et al. [31]).

7.4 Summary and conclusion

The groundwork for integrating a 2-axis accelerometer into a combined mobile phone/personal digital assistant was made. Software was written for the analysis and presentation of accelerometer data to the device operating system.

The goal of the project, to implement a gesture-based user interface on a handheld device, was not fulfilled, due to the fact that no working device was produced in time.

Future work involving 3-D sensing and rotation detection was briefly discussed. Possible applications concerning inertial sensing in handheld devices were presented.

Bibliography

[1] ARM Linux home page. `http://www.arm.linux.org.uk/`

[2] Stefan Hellkvist. On-line character recognition on small hand-held terminals using elastic matching. Master's thesis, Royal Institute of Technology, Stockholm 1999. Available online at `http://www.hellkvist.org/software/xmerlin/thesis.ps.gz`

[3] B. Myers, S. E. Hudson, and R. Pausch. Past, Present and Future of User Interface Software Tools. `http://www.cs.cmu.edu/~amulet/papers/futureofhci.pdf`

[4] C. Verplaetse. Inertial proprioceptive devices: Self-motion-sensing toys and tools. *IBM Systems Journal*, 35 (3&4), 1996.

[5] T. Marrin and J. Paradiso. The Digital Baton: a versatile performance instrument. *Proceedings of the International Computer Music Conference*, pp. 313-316, Computer Music Association, 1997.

[6] J. Paradiso. The Brain Opera technology: New Instruments and gestural sensors for musical interaction and performance.
Journal of New Music Research, 28(2) pp. 130-149, 1999

[7] Brain Opera Technical Systems site. `http://www.media.mit.edu/~joep/TTT.BO/index.html`

[8] H. Sawada and S. Hashimoto. Gesture Recognition using an accelerometer sensor and its application to musical performance control. *Electronics and Communications in Japan, Part 3*, 80(5) pp. 9-17, 1997.

[9] Jun Rekimoto. Tilting Operation for Small Screen Interfaces. *Proceedings of UIST'96*, pp. 167-168.

[10] J.F. Bartlett. Rock 'n' Scroll is here to stay. *IEEE Computer Graphics and Applications*, 20(3) pp. 40-45, May/June 2000.

[11] K. Hinckley, J. Pierce, M. Sinclair, and E. Horvitz. Sensing Techniques for Mobile Interaction. *Proceedings of the 13th annual ACM symposium on User Interface Software and Technology*, November 2000

[12] T. Marrin. Possibilities for the Digital Baton as a General-Purpose Gestural Interface. `http://www.acm.org/sigs/sigchi/chi97/proceedings/short-talk/tm.htm`

[13] K. Fishkin, A. Gujar, B. Harrison, T. Moran, and R. Want. Embodied User Interfaces for Really Direct Manipulation. *Communications of the ACM*, 43(9) pp. 75-80, September 2000.

[14] Ari Y. Benbasat. An Inertial Measurement Unit for User Interfaces. S.M. Thesis, Program in Media Arts and Sciences, Massachussetts Institute of Technology. September 2000

[15] Analog Devices. ADXL202 Data Sheet. `http://products.analog.com/products/info.asp?product=ADXL202`

[16] Microchip Technology Inc. PIC16C62X EPROM-Based 8-Bit CMOS Microcontroller. *Datasheet DS30235G*, Microchip Technology Inc., Chandler, Arizona. `ftp://ftp4.microchip.com/Download/Lit/PICmicro/16C62X/30235g.pdf`

[17] Microchip ProMate II programmer. `http://www.microchip.com/1010/pline/tools/picmicro/program/promate/index.htm`

[18] Microchip PICSTART Plus programmer. `http://www.microchip.com/1010/pline/tools/picmicro/program/picstart/index.htm`

[19] Cosmodog PIC Tools. `http://www.cosmodog.com/pic/index.html`

[20] Custom Computer Services, Inc. *PCB, PCM, and PCW PIC C Compiler Reference Manual*. November, 1998.

[21] Samuel P. Harbison, Guy L. Steele. *C, A Reference Manual*, 4th ed. Prentice-Hall, 1995 ISBN 0-13-326224-3

[22] Harvey Weinberg (1998). *Using the ADXL202 Duty Cycle Output*. `http://www.analog.com/techsupt/application_notes/ADXL202.pdf`

[23] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, Inc., 1998 ISBN 1-56592-292-1

[24] Andrea Controzzi. Linux PLIP mini-HOWTO. *Linux Documentation Project*, 1998 http://metalab.unc.edu/LDP/HOWTO/mini/PLIP.html

[25] Harvey Weinberg. *Embedding Temperature Information in the ADXL202's PWM Outputs.* http://www.analog.com/library/applicationNotes/dsp/pdf/202Embed.pdf

[26] Till Harbaum. Mulg II web page. http://bodotill.suburbia.com.au/mulg/mulg.html

[27] Kristof Van Laerhoven. Augmenting the iPAQ with Sensorboards. http://www.comp.lancs.ac.uk/~kristof/notes/ca_ipaq/index.html

[28] Palm Inc. http://www.palm.com/

[29] Windows Embedded home page. http://www.microsoft.com/windows/embedded/default.asp

[30] Compac iPAQ home page. http://www.compaq.com/showroom/handhelds.html

[31] S. Pradhan, C. Brignone, Jun-Hong Cui, A. McReynolds, M. Smith. *Websign: hyperlinks from a physical location to the web.* HP Laboratories Technical Report HPL-2001-140, 2001. http://www.hpl.hp.com/techreports/2001/HPL-2001-140.pdf

[32] Ari Y. Benbasat. iRX Project: Getting Started. http://www.media.mit.edu/~ayb/irx/GettingStarted/

Appendix A

Hardware and software requirements

As many of the things needed for this project have been mentioned sparingly, if at all, this is a list over the requirements in hardware and software needed for the project.

A.1 Hardware

- A PC or equivalent running Linux. The presence of a PC-style parallel port is necessary. This machine can be used both as a development machine and a controller for the device.
- A PC running Microsoft Windows. This is needed to control the PIC programmer. The machine has to have a working serial port.
- A Microchip Pro Mate II micro-controller device programmer.
- An adaptor board, model number `AC164010`, for the programming of PIC 16C622 devices.

A network between the two computers is a good idea. This means that files can be stored in a central location and accessed with the help of the SMB protocol, with the Linux machine acting as a server. Otherwise, FTP can be used.

The VNC (Virtual Network Computer) program is useful for communication between different operating systems. It allows the user to control two or more computers running a GUI from on physical location. It is available for free from `http://www.uk.research.att.com/vnc/`.

Of course, you also need the components listed in Figs. 2.1 and 2.2, along with wires, a power supply, and a soldering iron.

A.2 Software

- Linux, kernel version 2.x or higher. Any distribution will do. You need the complete kernel source and headers to compile the device driver.
- The ncurses console display library.
- Microsoft Windows. Versions are 3.1x and 95/98.
- Microchip MPLAB development environment, available for free download at `http://www.microchip.com/`.
- CCS C compiler. Details on purchase availability is at `http://www.ccsinfo.com/`.

A good tutorial on how to integrate the CCS compiler with MPLAB, along with setting up a project, is available at the iRX section of Ayi Y. Benbasat's homepage [32].

A.3 Colophon

This document was prepared with the teTeX distribution of LaTeX on a Debian/GNU Linux platform.

Images were created with Xfig and The Gimp.

Source and makefiles can be found at `http://stureby.net/gustaf/`.

Appendix B

Code listings

B.1 Micro-controller code

B.1.1 accel.c

```
/* accel.c
 * Standalone program for handling input from ADXL202 accelerometer
 * (c) Gustaf Erikson 1999
 * Includes code by Stefan Hellkvist
 *
 */

#include <16C622.H>             /* Standard header file for the PIC 16C622  */
#include "math32.c"             /* math routines */
#include "counter.c"            /* Stefan's counter code */

/* comment out this line to get production code: */
#define TEST
/* Options for the PIC:
   HS = high-speed crystal oscillator
   No code protection
   No watchdog timer
   No brownout protection
   No power-up timer
*/
#fuses HS, NOPROTECT, NOWDT, NOBROWNOUT, NOPUT

#ifdef TEST
#define TKCKL 1
#define TKCKH 1
#define MDELAY delay_ms
#else
#define TKCKL 25                /* defines how long the flank should be */
#define TKCKH 25
#define MDELAY delay_us
#endif

#define MSDATA PIN_B4           /* input pins on the ARM */
#define MSCLK PIN_B3

#define ACCEL_X PIN_B0          /* Input pins on the PIC */
```

```
#define ACCEL_Y PIN_B1
/* A packet will be sent if some value
   has changed more than the limit. */
#define X_CHANGE_LIMIT 0
#define Y_CHANGE_LIMIT 0
#define MAX_WAIT 65000

/* Defines the direction of the ports
   A 1 defines input, 0 output
   0x85 = 0b10000101
   0x86 = 0b10000110 */
#byte tris_a = 0x85
#byte tris_b = 0x86

#byte cmcon = 0x1f
#byte vrcon = 0x9f

#byte tmr0 = 0x01               /* timer port setup */



#use delay(clock=12000000)      /* the outer clock is 12 Mhz */

void initCom()                  /* initialize communications */
{
        /* Set RB4, RB3 as outputs, all others are
           inputs. */
        set_tris_b( 0xe7 );
        output_high( MSCLK );
        output_high( MSDATA );
}
int isOddParity( char b )       /* Check parity -- needed for the ARM */
{
        int ones = 0, i;

        for ( i = 0; i < 8; i++ )
                if ( bit_test( b, i ) )
                        ones++;
        return ones % 2;
}
/*
 * writeByte
 * Writes a byte on the ps/2 lines to the StrongARM.
 * RB4 is MSDATA and RB3 is MSCLK
 * arg b: the byte to write
 */

void writeByte( char b )
{
        int i;

        // pull data low
        output_low( MSDATA );
        MDELAY( TKCKL );
        output_low( MSCLK );
        MDELAY( TKCKL );

        for ( i = 0; i < 8; i++ )
        {
                output_high( MSCLK );
                // set valid data on pin
                if ( bit_test( b, i ) )
```

```
                output_high( MSDATA );
        else
                output_low( MSDATA );
        MDELAY( TKCKH );

        // clock the data with a negative flank
        output_low( MSCLK );
        MDELAY( TKCKL );
    }

    // send parity bit (odd parity)
    output_high( MSCLK );
    if ( isOddParity( b ) )
            output_high( MSDATA );
    else
            output_low( MSDATA );
    MDELAY( TKCKH );
    output_low( MSCLK ); // clock parity data with negative flank
    MDELAY( TKCKL );

    // send stop bit
    output_high( MSCLK );
    output_high( MSDATA );
    MDELAY( TKCKL );
    output_low( MSCLK ); // clock stop bit with negative flank
    MDELAY( TKCKL );

    // set clock high and everything is ready for next byte
    output_high( MSCLK );
    MDELAY( TKCKL );
}

void output_acc(long int x, long int y)
{
    char b = 0xc0;          /* this value defines the header for
                               the accelerometer*/

    /* Make the accel. values into 14-bit
       integers, the two upper bits will always
       be zero. */
    x = x & 0x3fff;
    y = y  & 0x3fff;
    /* write header */
    writeByte( b );
    b = x & 0x007f;        /* x: low 7 bits  */
    writeByte( b );
    b = (x & 0x3f80) >> 7; /* x: high 7 bits */
    writeByte( b );
    b = y & 0x007f;        /* y: low 7 bits  */
    writeByte( b );
    b = (y & 0x3f80) >> 7; /* y: high 7 bits */
    writeByte( b );
                           /* two dummy z:s */
    writeByte( 0 );
    writeByte( 0 );
}

void get_times(long int *tb, long int *tc, long int *td)

/* This function reads the accelerometer pins and counts the length of
   the pulses on them. The result is returned for processing. */
{
```

```
        /*  result.t_a = 0; */
        /*  wait_for_low_to_high_X; */
        while(input( ACCEL_X ));        /* if high, wait for low */
        delay_us(3);                    /* account for fall time */
        while(!input( ACCEL_X ));       /* wait for new high */
        startTimer();
        /*  wait_for_low_X(); */
        delay_us(3);                    /* account for rise time */
        while(input( ACCEL_X ));
        *tb = getTicks();

        /*  wait_for_low_to_high_Y(); */
        while(input( ACCEL_Y ));        /* if high, wait for low */
        delay_us(3);                    /* account for fall time */
        while(!input( ACCEL_Y ));       /* wait for new high */
        *tc = getTicks();
        /*  wait_for_low_Y(); */
        delay_us(3);                    /* account for rise time */
        while(input( ACCEL_Y ));
        *td = getTicks();
}

int main()
{                               /* struct long32 is defined in math32.c
                                   long int is 16 bits
                                 */
        struct long32 K, working, T2cal_32bit, T2_32bit, tmp_32;
        long int Zactual_x, Zactual_y, Zcal_x, Zcal_y, T2cal;
        long int t_b, t_c, t_d, T2;
        long int T1_x, T1_y;
        long int acc_x, acc_y;
        long int bit_scale_factor;
        long diff_x, diff_y;

        /* initialization */
        initTimer();
        initCom();
        delay_ms( 1000 );       /* wait a second for things to settle down */

        /* Calibration start
           This calibration depends on the device being level during it.*/

        bit_scale_factor = 0x100; /* bit_scale_factor = 256 */

        get_times(&t_b, &t_c, &t_d);
        Zcal_x = t_b;
        Zcal_y = t_d - t_c;
        T2cal = (t_d - (t_d - t_c)/2) - t_b/2;

        T2cal_32bit.hi = 0x0;
        T2cal_32bit.lo = T2cal;
        working.hi = 0x0;
        working.lo = bit_scale_factor;
        mul32(&working, &T2cal_32bit);  /* working is now
                                           T2cal*b_s_f */
        tmp_32.hi = 0x0;
        tmp_32.lo = 0x4;
        mul32(&working, &tmp_32);  /* working is now 4*working */
        div32(&working, &T2cal_32bit);
        K = working;            /* Scale factor K = 4*(T2cal*BSF)/T2cal */
```

```
/* calibration end */
                                /* Main loop */
        do {
                get_times(&t_b, &t_c, &t_d);    /* calculate t2 */
                T1_x = (t_b);
                T1_y = (t_d - t_c);
                T2 = (t_d - (t_d - t_c)/2) - t_b/2;

                T2_32bit.hi = 0x0;      /* Zactual = (Zcal * T2)/T2cal; */
                T2_32bit.lo = T2;
                working.hi = 0x0;
                working.lo = Zcal_x;
                mul32(&working, &T2_32bit);
                div32(&working, &T2cal_32bit);
                Zactual_x = working.lo;

                working.hi = 0x0;
                working.lo = Zcal_y;
                mul32(&working, &T2_32bit);
                div32(&working, &T2cal_32bit);
                Zactual_y = working.lo;

                                /* calculate ax, ay */
                                /* acc = (K * (T1 - Zactual)) / T2actual */
                diff_x = (T1_x - Zactual_x);
                                /* check for sign -- if highest bit of
                                   diff_x is set, the value is
                                   negative. In that case, set
                                   working.hi to 0xffff*/
                if ( 0x8000 & diff_x ) working.hi = 0xffff;
                else working.hi = 0x0;

                working.lo = diff_x;
                mul32(&working, &K);
                div32(&working, &T2_32bit);
                acc_x = working.lo;

                diff_y = (T1_y - Zactual_y);
                                /* check for sign */
                if ( 0x8000 & diff_y ) working.hi = 0xffff;
                else working.hi = 0x0;

                working.lo = diff_y;
                mul32(&working, &K);

                div32(&working, &T2_32bit);
                acc_y = working.lo;

                                /* pack into packets */
                output_acc(acc_x, acc_y);
        } while (1);

        return (0);
}
```

B.1.2 math32.c

```
/* math32.c
   Routines for 32 bit integer multiplication and division.
   From the file 'math.c' in the CCS distribution
   (C) Copyright 1996,1997 Custom Computer Services
   Published with permission.
*/

struct long32 {unsigned long lo;  long hi;};

void mul32(struct long32 *a, struct long32 *b) {
                        // a = a*b
   byte r[4], c[4], sign, count;

   r[0] = *(&a->lo);
   r[1] = *(&a->lo + 1);
   r[2] = *(&a->lo + 2);
   r[3] = *(&a->lo + 3);

   c[0] = *(&b->lo);
   c[1] = *(&b->lo + 1);
   c[2] = *(&b->lo + 2);
   c[3] = *(&b->lo + 3);

   a->lo = 0;
   a->hi = 0;

#asm
   movlw    31
   movwf    count

   movf     r[3],w
   movwf    sign
   movf     c[3],w
   xorwf    sign,f

   btfss    r[3],7
   goto     chkb
   comf     r[0],f
   comf     r[1],f
   comf     r[2],f
   comf     r[3],f
   incfsz   r[0],f
   goto     chkb
   incfsz   r[1],f
   goto     chkb
   incfsz   r[2],f
   goto     chkb
   incf     r[3],f

chkb:
   btfss    c[3],7
   goto     loop
   comf     c[0],f
   comf     c[1],f
   comf     c[2],f
   comf     c[3],f
   incfsz   c[0],f
   goto     loop
   incfsz   c[1],f
   goto     loop
```

```
   incfsz   c[2],f
   goto     loop
   incf     c[3],f

loop:
   btfss    r[0],0
   goto     shft
   movf     a,w
   movwf    4
   movf     c[0],w
   addwf    0,f
   movf     c[1],w
   incf     4,f
   btfsc    3,0
   incfsz   c[1],w
   addwf    0,f
   movf     c[2],w
   incf     4,f
   btfsc    3,0
   incfsz   c[2],w
   addwf    0,f
   movf     c[3],w
   incf     4,f
   btfsc    3,0
   incfsz   c[3],w
   addwf    0,f

shft:
   bcf      3,0
   rrf      r[3],f
   rrf      r[2],f
   rrf      r[1],f
   rrf      r[0],f

   bcf      3,0
   rlf      c[0],f
   rlf      c[1],f
   rlf      c[2],f
   rlf      c[3],f
   decfsz   count,f
   goto     loop

   movf     a,w
   addlw    3
   movwf    4
   bcf      0,7
   btfss    sign,7
   goto     end

   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
   comf     0,f
   incfsz   0,f
   goto     end
   incf     4,f
   incfsz   0,f
   goto     end
   incf     4,f
```

```
   incfsz   0,f
   goto     end
   incf     4,f
   incf     0,f
end:
   nop
#endasm
}
void div32(struct long32 *a,struct long32  *b) {
                        // a = a/b
   byte c[4], d[4], sign, count, negb;

   c[0] = *(&a->lo);
   c[1] = *(&a->lo + 1);
   c[2] = *(&a->lo + 2);
   c[3] = *(&a->lo + 3);

   d[0] = 0;
   d[1] = 0;
   d[2] = 0;
   d[3] = 0;

   a->lo = 0;
   a->hi = 0;
   count = 32;
   negb = 0;

#asm
   movf     c[3],w
   movwf    sign
   movf     b,w
   addlw    3
   movwf    4
   movf     0,w
   xorwf    sign,f

   btfss    c[3],7
   goto     chkb
   comf     c[0],f
   comf     c[1],f
   comf     c[2],f
   comf     c[3],f
   incfsz   c[0],f
   goto     chkb
   incfsz   c[1],f
   goto     chkb
   incfsz   c[2],f
   goto     chkb
   incf     c[3],f

chkb:
   movf     b,w
   addlw    3
   movwf    4
   btfss    0,7
   goto     loop
   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
```

```
   comf     0,f
   incfsz   0,f
   goto     loop
   incf     4,f
   incfsz   0,f
   goto     loop
   incf     4,f
   incfsz   0,f
   goto     loop
   incf     4,f
   incf     0,f
   bsf      negb,0

loop:
   bcf      3,0
   rlf      c[0],f
   rlf      c[1],f
   rlf      c[2],f
   rlf      c[3],f
   rlf      d[0],f
   rlf      d[1],f
   rlf      d[2],f
   rlf      d[3],f

   movf     b,w
   addlw    3
   movwf    4
   movf     0,w
   subwf    d[3],w
   btfss    3,2
   goto     chk
   decf     4,f
   movf     0,w
   subwf    d[2],w
   btfss    3,2
   goto     chk
   decf     4,f
   movf     0,w
   subwf    d[1],w
   btfss    3,2
   goto     chk
   decf     4,f
   movf     0,w
   subwf    d[0],w

chk:
   btfss    3,0
   goto     skip
   movf     b,w
   movwf    4
   movf     0,w
   subwf    d[0],f
   incf     4,f
   movf     0,w
   btfss    3,0
   incfsz   0,w
   subwf    d[1],f
   incf     4,f
   movf     0,w
   btfss    3,0
   incfsz   0,w
   subwf    d[2],f
```

```
   incf     4,f
   movf     0,w
   btfss    3,0
   incfsz   0,w
   subwf    d[3],f
   bsf      3,0

skip:
   movf     a,w
   movwf    4
   rlf      0,f
   incf     4,f
   rlf      0,f
   incf     4,f
   rlf      0,f
   incf     4,f
   rlf      0,f
   decfsz   count,f
   goto     loop

   movf     a,w
   addlw    3
   movwf    4
   bcf      0,7
   btfss    sign,7
   goto     retb

   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
   comf     0,f
   incfsz   0,f
   goto     retb
   incf     4,f
   incfsz   0,f
   goto     retb
   incf     4,f
   incfsz   0,f
   goto     retb
   incf     4,f
   incf     0,f

retb:
   btfss    negb,0
   goto     end
   movf     b,w
   addlw    3
   movwf    4
   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
   comf     0,f
   decf     4,f
   comf     0,f
   incfsz   0,f
   goto     end
   incf     4,f
   incfsz   0,f
```

```
   goto     end
   incf     4,f
   incfsz   0,f
   goto     end
   incf     4,f
   incf     0,f
end:
   nop
#endasm
}
```

B.1.3 counter.c

```
/* Common counter routines for the PIC controlling the touchscreen and
   accelermeter.
   (c) Stefan Hellkvist 1999 */

#byte option = 0x81
static long count;

#int_rtcc                       /* Interrupt handler */

void rtcc_handler()             /* This function adds a count of 256
                                   when the interrupt counter
                                   overflows */
{
        count += 256;
}

void resetTimer()               /* Reset the timer */
{
        set_rtcc( 0 );
        count = 0;
}

void initTimer()                /* Use the internal real-time counter,
                                   set the prescaler to 2 (only count
                                   every second interrupt). */
{
        setup_counters( RTCC_INTERNAL, RTCC_DIV_2 );
}

void startTimer()               /* Enable the interrupt counter, set
                                   count to zero */
{
        enable_interrupts( GLOBAL );
        enable_interrupts( INT_RTCC );
        resetTimer();
}

void stopTimer()                /* Stop the counter */
{
        disable_interrupts( INT_RTCC );
}

long getTicks()         /* Return the saved counts plus the
                                   value of the counter when it was
                                   stopped */
{
        return count + (long) get_rtcc();
}
```

B.2 Multi-axis device code

B.2.1 multi_acc.c

```
/* multi_acc.c
 code for 3 accelerometers
 (c) Gustaf Erikson 1999
*/

#include <16C622.H>
#include "math32.c"             /* math routines */
#include "counter.c"            /* Stefan's counter code */

/* comment out this line to get production code: */
#define TEST

#fuses HS, NOPROTECT, NOWDT, NOBROWNOUT, NOPUT

#ifdef TEST
#define TKCKL 1
#define TKCKH 1
#define MDELAY delay_ms
#else
#define TKCKL 25                /* defines how long the flank should be */
#define TKCKH 25
#define MDELAY delay_us
#endif

#define MSDATA PIN_B4           /* input pins on the ARM */
#define MSCLK PIN_B3

//#define ACCEL_X PIN_B0                /* Input pins on the PIC */
//#define ACCEL_Y PIN_B1

/* Input pins for the different accelerometers:
   id  X_PORT   Y_PORT

   0   PIN_B0   PIN_B1
   1   PIN_A0   PIN_A1
   2   PIN_A2   PIN_A3
*/
                                /* A packet will be sent if some value
                                has changed more than the limit. */
#define X_CHANGE_LIMIT 0
#define Y_CHANGE_LIMIT 0
#define MAX_WAIT 65000

#byte tris_a = 0x85
#byte tris_b = 0x86
#byte cmcon = 0x1f
#byte tmr0 = 0x01

#byte vrcon = 0x9f

#use delay(clock=12000000)      /* the outer clock is 12 Mhz */

void initCom()                  /* initialize communications */
{
                                /* Set RB4, RB3 as outputs, all others are
                                   inputs. */
        set_tris_b( 0xe7 );
```

```
        output_high( PIN_B3 );
        output_high( PIN_B4 );
}
int isOddParity( char b )
{
        int ones = 0, i;

        for ( i = 0; i < 8; i++ )
                if ( bit_test( b, i ) )
                        ones++;
        return ones % 2;
}
/*
 * writeByte
 * Writes a byte on the ps/2 lines to the StrongARM.
 * RB4 is MSDATA and RB3 is MSCLK
 * arg b: the byte to write
 */

void writeByte( char b )
{
        int i;

        // pull data low
        output_low( MSDATA );
        MDELAY( TKCKL );
        output_low( MSCLK );
        MDELAY( TKCKL );

        for ( i = 0; i < 8; i++ )
        {
                output_high( MSCLK );
                // set valid data on pin
                if ( bit_test( b, i ) )
                        output_high( MSDATA );
                else
                        output_low( MSDATA );
                MDELAY( TKCKH );

                // clock the data with a negative flank
                output_low( MSCLK );
                MDELAY( TKCKL );
        }

        // send parity bit (odd parity)
        output_high( MSCLK );
        if ( isOddParity( b ) )
                output_high( MSDATA );
        else
                output_low( MSDATA );
        MDELAY( TKCKH );
        output_low( MSCLK ); // clock parity data with negative flank
        MDELAY( TKCKL );

        // send stop bit
        output_high( MSCLK );
        output_high( MSDATA );
        MDELAY( TKCKL );
        output_low( MSCLK ); // clock stop bit with negative flank
        MDELAY( TKCKL );

        // set clock high and everything is ready for next byte
```

```
        output_high( MSCLK );
        MDELAY( TKCKL );
}

void output_acc(int id, long int x, long int y)
{
        char b;
        switch( id ){           /* arrange status byte for output */
        case 0: b = 0x80; break; /* 0b10000000 */
        case 1: b = 0xa0; break; /* 0b10100000 */
        case 2: b = 0xc0; break; /* 0b11000000 */
        };
        /* Make the accel. values into 14-bit
           integers, the two upper bits will always
           be zero. */

        x = x & 0x3fff;
        y = y  & 0x3fff;
        /* write header */
        writeByte( b );
        b = x & 0x007f;        /* x: low 7 bits  */
        writeByte( b );
        b = (x & 0x3f80) >> 7; /* x: high 7 bits */
        writeByte( b );
        b = y & 0x007f;        /* y: low 7 bits  */
        writeByte( b );
        b = (y & 0x3f80) >> 7; /* y: high 7 bits */
        writeByte( b );
        /* two dummy z:s */
        writeByte( 0 );
        writeByte( 0 );
}
void get_ports(int id, int *x_port, int *y_port)
{
        switch(id){
        case 0:
                *x_port = PIN_B0;
                *y_port = PIN_B1;
                break;
        case 1:
                *x_port = PIN_A0;
                *y_port = PIN_A1;
                break;
        case 2:
                *x_port = PIN_A2;
                *y_port = PIN_A3;
                break;
        };
}

void get_times(int id, long int *tb, long int *tc, long int *td)
{
        get_ports( id, &accel_x, &accel_y );
        /*  result.t_a = 0; */
        /*  wait_for_low_to_high_X; */
        while(input( accel_x ));        /* if high, wait for low */
        delay_us(3);                    /* account for fall time */
        while(!input( accel_x ));       /* wait for new high */
        startTimer();
        /*  wait_for_low_X(); */
        delay_us(3);                    /* account for rise time */
        while(input( accel_x ));
```

```
        *tb = getTicks();

        /*  wait_for_low_to_high_Y(); */
        while(input( accel_y ));        /* if high, wait for low */
        delay_us(3);                    /* account for fall time */
        while(!input( accel_y ));       /* wait for new high */
        *tc = getTicks();
        /*  wait_for_low_Y(); */
        delay_us(3);                    /* account for rise time */
        while(input( accel_y ));
        *td = getTicks();
}

int main()
{
        struct long32 working, T2_32bit, tmp_32;
        struct long32 K[3], T2cal_32bit[3];
        long int Zactual_x, Zactual_y;
        long int Zcal_x[3], Zcal_y[3];
        long int T2cal;
        long int t_b, t_c, t_d, T2;
        long int T1_x, T1_y;
        long int acc_x, acc_y;
        long int test_hi, test_lo;
        long int bit_scale_factor;
        long diff_x, diff_y;
        int id;                 /* which accelerometer to work on */
        /* initialization */
        initTimer();
        initCom();
        delay_ms( 1000 );       /* wait a second for things to settle down */
        /* calibration(&T2cal, &Zcal_x, &Zcal_y, &bit_scale_factor, &K); */

/* Calibration start */

        bit_scale_factor = 0x100; /* bit_scale_factor = 256 */
        for (id=0; id < 3; id++){ /* perform cal for each meter in turn */
                get_times( id, &t_b, &t_c, &t_d );
                                /* every acc.meter needs own:
                                   Zcal_x, Zcal_y
                                   K, T2cal_32bit */
                Zcal_x[id] = t_b;
                Zcal_y[id] = t_d - t_c;
                T2cal = (t_d - (t_d - t_c)/2) - t_b/2;
                /* K = (4 * (T2cal * bit_scale_factor ))/ T2cal; */
                T2cal_32bit[id].hi = 0x0;
                T2cal_32bit[id].lo = T2cal;
                working.hi = 0x0;
                working.lo = bit_scale_factor;
                mul32(&working, &T2cal_32bit);  /* working is now
                                                   T2cal*b_s_f */
                tmp_32.hi = 0x0;
                tmp_32.lo = 0x4;
                mul32(&working, &tmp_32);  /* working is now 4*working */
                div32(&working, &T2cal_32bit);
                K[id] = working;
        }

/* Calibration end */

        do {
                for (id=0; id<3; id++){ /* go thru each meter and calculate */
```

```
                        get_times(id, &t_b, &t_c, &t_d);/* calculate t2 */
                        T1_x = (t_b);
                        T1_y = (t_d - t_c);
                        T2 = (t_d - (t_d - t_c)/2) - t_b/2;

                        T2_32bit.hi = 0x0;  /* Zactual = (Zcal * T2)/T2cal; */
                        T2_32bit.lo = T2;
                        working.hi = 0x0;
                        working.lo = Zcal_x[id];
                        mul32(&working, &T2_32bit);
                        div32(&working, &T2cal_32bit[id]);
                        Zactual_x = working.lo;

                        working.hi = 0x0;
                        working.lo = Zcal_y[id];
                        mul32(&working, &T2_32bit);
                        div32(&working, &T2cal_32bit[id]);
                        Zactual_y = working.lo;

                                /* calculate ax, ay */
                                /* acc = (K * (T1 - Zactual)) / T2actual */
                        diff_x = (T1_x - Zactual_x);
                                /* check for sign */
                        if ( 0x8000 & diff_x ) working.hi = 0xffff;
                        else working.hi = 0x0;

                        working.lo = diff_x;
                        mul32(&working, &K[id]);
                        div32(&working, &T2_32bit);
                        acc_x = working.lo;

                        diff_y = (T1_y - Zactual_y);
                                /* check for sign */
                        if ( 0x8000 & diff_y ) working.hi = 0xffff;
                        else working.hi = 0x0;

                        working.lo = diff_y;
                        mul32(&working, &K[id]);

                        div32(&working, &T2_32bit);
                        acc_y = working.lo;

                                /* pack into packets */
                        output_acc(id, acc_x, acc_y);
                }
        } while (1);

        return (0);
}
```

B.3 Linux code

B.3.1 picreader.c

```
/* picreader.c
 * Device driver for reading the input from a Tifon test board over the
 * parallel port.
 *
 * compilation: 'gcc -D__KERNEL__ -DMODULE -O -Wall -c  picreader.c'
 *
 * installation: copy the resulting picreader.o to
 *   /lib/modules/<kernel version>/misc
 *
 * Load with 'insmod picreader'
 * Unload with 'rmmod picreader'
 *
 * (c) Gustaf Erikson and Stefan Hellkvist 1999
 */


#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/tqueue.h>
#include <linux/sched.h>
#include <linux/ioport.h>
#include <linux/malloc.h>
#include <asm/io.h>

#define BASEPORT 0x3BC /* This defines the parallel port, specifically
                          lp1 */

#define IRQ 7                   /* lp1 has interrupt request nr. 7 */
#define TEST_CLK(x) (0x10 & x)
#define TEST_DATA(x) (0x20 & x)

#define BUFFER_SIZE 100


static unsigned char cur_byte = 0, bit = 0, cur_parity;
static unsigned char pic_buffer[BUFFER_SIZE];
static int first = 0, last = 0;
static struct wait_queue *wq = NULL;

                                /* The central part of the module.
                                This will be invoked when there is a
                                signal on the interrupt pin. The
                                signal will be the clock byte. */

static void intrpt_routine(int not, void *not_u, struct pt_regs *not_used)
{
  static int started = 0;
  unsigned char status = inb(BASEPORT + 1);
  // printk(KERN_DEBUG "interrupt");
  if (!started)                 /* Check for positive flank on data pin*/
    {
      if (!TEST_DATA(status))
        started = 1;
      return;
    }
```

```
  if (bit < 8)                  /* Read data */
    {
      if (TEST_DATA(status))
        cur_byte |= 1 << bit;   /* OR it to the current byte */
      bit++;                    /* increment bit count */
    }
  else
    if (bit == 8)               /* Read parity bit (used for
                                   compatibility with the StrongARM) */
      {
        cur_parity = TEST_DATA(status);
        bit++;
      }
    else
      if (bit == 9)             /* Read stop bit */
        {
          pic_buffer[last] = cur_byte; /* Put the byte into the buffer */
          last = (last + 1) % BUFFER_SIZE; /* If the buffer is full,
                                              start over */
          bit = 0;              /* Start over */
          cur_byte = 0;
          wake_up_interruptible(&wq);
          started = 0;
        }
}

/* These functions are needed for character devices */

                                /* Open the PIC reader */
int pic_open(struct inode *ip, struct file *filp)
{
        /* printk(KERN_INFO "picreader: device opened\n"); */
        /* MOD_INC_USE_COUNT; <-- This function increases the use
 count of devices. It is commented out due to errors during
 testing. */
  return 0;
}

void pic_release(struct inode *ip, struct file *filp)
{
        /* printk(KERN_INFO "picreader: device released\n"); */
        /* MOD_DEC_USE_COUNT; <-- same problem as above */
}

                                /* Data will not be written to the
                                   device, but this function is
                                   included for completeness */
int pic_write(struct inode *ip, struct file *filp, const char *buf, int count)
{
  return 0;
}

int pic_read(struct inode *ip, struct file *filp, char *buf, int count)
{
  int c = 0;
  while ( last == first )       /* If there is no data, wait until
                                   there is */
    {
      interruptible_sleep_on(&wq);
      if (current->signal & ~current->blocked)
        return -ERESTARTSYS;
    }
```

```
  while ( last != first && c < count) /* Read the data from the pic
                                         and copy it to the device
                                         file */
    {
      memcpy_tofs(buf + c, pic_buffer + first, 1);
      first = (first + 1) % BUFFER_SIZE;
      c++;
    }
  return c;
}

                                /* Define the file operations for the
                                   device */
struct file_operations pic_fops =
{
  NULL,
  pic_read,
  pic_write,
  NULL,
  NULL,
  NULL,
  NULL,
  pic_open,
  pic_release,
};


#define PIC_MAJOR 100
                                /* initiate module */
int init_module()
{
  int result = register_chrdev(PIC_MAJOR, "picreader", &pic_fops);
  if (result < 0)
    {
      printk(KERN_WARNING "picreader: can't get major %d\n", PIC_MAJOR);
      return result;
    }

  result = check_region(BASEPORT, 4);
  if (result)
    {
      printk(KERN_WARNING "picreader: can't get I/O address\n");
      return result;
    }
  request_region(BASEPORT, 4, "picreader");
                                /* Request an interrupt, and execute
                                   intrp_routine() when we get it */
  result = request_irq(IRQ, intrpt_routine, SA_INTERRUPT, "picreader",
                       NULL);
  if (result)
    printk(KERN_INFO "picreader: can't get irq %i\n", IRQ);
  else
    outb(0x10, BASEPORT + 2); // enable interrupt


  printk(KERN_INFO "picreader: init_module\n");

  return 0;
}
                                /* cleanup code */
void cleanup_module()
```

```
{
  unregister_chrdev(PIC_MAJOR, "picreader");
  free_irq(IRQ, NULL);
  release_region(BASEPORT, 4);
  printk(KERN_INFO "picreader: cleanup_module\n");
}
```

B.3.2 acc_display.c

```
/* Code for displaying accelerometer output
 * Compile with
 *         gcc -O -Wall -lcurses -o acc_display acc_display.c
 *
 * (c) Gustaf Erikson 1999
 */

#include <stdio.h>
#include <curses.h>

struct packet
{
        unsigned char status;
        unsigned char data[6];
};

void
readpacket( struct packet *p )
{
        unsigned char c;
        while (!((c = getchar()) & 0x80)); /* wait for status byte */
        p->status = c;
        for ( c = 0; c < 6; c++)
                p->data[c] = getchar();
}

                                /* Convert packet values to 16-bit
                                   integer */
static int
toInt(unsigned char low, unsigned char high)
{
        return low + (((int) high) << 7);
}
                                /* Check for sign and convert to
                                   [-127, 127] range*/
int check_sign(int raw)
{
        int sign_value = 0x2000;
        int max_value = 0x4000;
        if ( raw & sign_value ){
                raw = raw - max_value;
        }
        return raw;
}

void
printpacket(struct packet *p)   /* Nice formatting of output with
                                   curses */
{

        move(0, 0); refresh();
        printw("status: %x\n", p->status);

        move(1, 0); refresh();
        printw("x:      %+d\n", check_sign(toInt(p->data[0], p->data[1])));
        move(2, 0); refresh();
        printw("y:      %+d\n", check_sign(toInt(p->data[2], p->data[3])));
        move(3, 0); refresh();
        printw("z:      %+d\n", check_sign(toInt(p->data[4], p->data[5])));
}
```

```
int
main()
{
        struct packet p;
        initscr();              /* Initiate screen for display */
        while (1)
        {
                readpacket(&p);
                printpacket(&p);
        }
        return 0;
}
```