AN-900 APPLICATION NOTE

**One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • [www.analog.com](http://www.analog.com/)**

**使用单个加速度计增强计步器的性能, 原文链接:**

**http://notes-application.abcelectronique.com/013/13-14983.pdf**

**Enhancing the Performance of Pedometers Using a Single Accelerometer**

**by Jim Scarlett**

The AN-602 application note examined the use of an Analog Devices, Inc. accelerometer to make a simple but relatively accurate pedometer. However, newer devices have been introduced that allow the use of accelerometers in more cost- sensitive applications. Thus, applications such as pedometers are finding themselves in many consumer devices such as cellular handsets.

Given this trend, a closer examination was made of pedome- ters using a single accelerometer. The AN-602 technique was implemented in an attempt to duplicate its results. Though the algorithm performed well, the same accuracy was not duplicated. In particular, there was greater variation than expected from person to person, as well as when one person used a different pace and stride length. This led to an investigation of potential improvements to the algorithm used in AN-602.

The tests were done using two different pedometer test boards, both utilizing an ADuC7020 ARM7® controller. One setup was a combination of the ADuC7020 microcontroller and ADXL322 accelerometer evaluation boards, with an added 16 × 2 char- acter LCD display. The other was a custom board using the ADuC7020 and an ADXL330 3-axis accelerometer, again with

a 16 × 2 character LCD display. See [Figure 5](#_bookmark4) for the custom board schematic.

#### AN-602 METHOD

The underlying reasoning in the AN-602 method is based on the principle that the vertical bounce in an individual’s step is directly correlated to that person’s stride length (see [Figure 1](#_bookmark0)).

Because the angles α and θ are equal, the stride can be shown to be a multiple of the maximum vertical displacement. Differences in an individual’s leg length are accounted for, given that for the same angles the vertical displacement would be greater or smaller for taller or shorter people.

Using an accelerometer, however, gives changes in acceleration rather than displacement. These acceleration measurements must be converted to a distance before they can be used. With the limited computing power available in the AN-602 setup, a simple formula was used to approximate the double integral needed for the conversion. With plenty of processing power available, an attempt was made to do the discrete integrals directly for this experiment.

A simple method was chosen to calculate the integrals. After each step was determined, all of the acceleration samples within that step were added to obtain a set of velocity samples. The velocity samples for each step were normalized such that the final sample was zero. They were then added together to get a value for the displacement.

This technique looked promising initially, because measured distances were relatively consistent for one subject walking

a course multiple times. However, the variance problem from person to person was exacerbated, as was the variance for one subject at different paces. This led to an investigation of whether the problem is with the model itself.

**HIP**

**BOUNCE**

**α**

**θ**

**RIGHT LEG**

**LEFT LEG**

**LEFT LEG**

**RIGHT LEG**

*Figure 1. Vertical Movement of Hip While Walking*

06617-001

**AN-900**

#### UNDERSTANDING THE MODEL

This model has two primary conditions. First, it assumes that the foot actually makes contact with the ground at a single point. Second, it assumes that the impact of each foot on the ground is perfectly elastic. Of course, neither of these is the case. The question arose of whether this can explain the large variations encountered. Based on this experiment, it is safe to state that it does explain much of the variation.

To understand this, it helps to look at the measured acceleration values over several steps, shown in [Figure 2](#_bookmark2). Different sources of spring in one person’s step are shown on the data.

**HEEL-TO-BALL BALL-TO-TOE** **TOE UP**

**500**

**450**



**EL**

**ACC**

**400**

**350**

**300**

**250**

**200**

**150**

**100**

**50**

**0**

**1**

**15 29 43 57 71 85 99 113 127 141 155 169**

06617-003

**ACCELERATION**

**340**

**290**

**240**

**190**

**140**

**1**

**15 29**

06617-002

**HEEL DOWN 43 57 71**

**85 99**

**ACCEL 113 127 141 155 169**

*Figure 3. Acceleration Graph for Subject 1 at a Fast Pace*

There are some important differences between the two plots. The bottom part of the curve for each step is slightly narrower in [Figure 3](#_bookmark1) and the tops of the curves are more consistent (fewer distinctive peaks). These differences result in a higher average value of the samples compared to the minimum and maximum sample values.

For comparison purposes, review the data plot for a different individual in [Figure 4](#_bookmark3). The stride length is very similar to that of Subject 1 in [Figure 2](#_bookmark2). However, the data itself looks very different.

**400**



**SERIES 1**

**ACCELLERATION**

*Figure 2. Acceleration Graph for Subject 1 at a Normal Pace*

[Figure 2](#_bookmark2) demonstrates the problems encountered trying to translate acceleration measurements into an accurate distance calculation. Methods that take into account the peak-to-peak change (or even those that integrate the data) run into trouble with this type of data. The cause of this difficulty is the variation in spring in the steps of different people, or in the steps of one person using different paces from one measurement to another.

[Figure 3](#_bookmark1) shows the same subject with a longer and faster stride. The peak-to-peak acceleration difference is larger, and the various spring points look different. Thus, the amount of data representing spring data vs. the amount representing real data is

**350**

**300**

**250**

**200**

**150**

**100**

**50**

**1**

**15 29 43 57 71 85 99 113 127 141 155 169**

06617-004

**ACCELERATION**

different compared with [Figure 2](#_bookmark2). However, the algorithm only sees a set of acceleration measurements and does not note the context of those measurements. The problem, therefore, is how to remove the effect of the spring in a subject’s step without removing useful data.

*Figure 4. Acceleration Graph for Subject 2 at a Normal Pace*

The stride of Subject 2 has a great deal more spring in it than that of Subject 1 (shown in [Figure 2](#_bookmark2)). Yet both sets of data represent roughly the same distance walked. Calculating distance solely on the peak values gives widely varying results. Using a simple double integration suffers from the same problem.

Rev. 0 | Page 2 of 16

**AN-900**

#### SOLVING THE PROBLEM OF SPRING

All efforts to come up with a decent solution to this problem using straightforward calculations had the same problems. This led to a series of attempts to normalize the data in a way that eliminated the spring, but all these attempts proved unsuc- cessful. The main reason seemed to be that they required some

The ratio used in the last test seemed to reflect the differences in the spring of different subjects’ steps. It made sense to try combining the two methods examined here. Going back to the original idea of using a double integral, this ratio was used

as a correction factor to remove the spring data from the calcu- lation. The resulting formula is therefore

knowledge of the context of the data. In actual use, however, the system has no idea what is going on outside; all it has are data points. The solution needs to be able to operate on the data without context.

A possible solution to this problem began to emerge. It was

*d*  *k* 

where *accel* represents all measured acceleration values for the step.

(*max*  *min*) \* (*accel*  *avg*)

(*avg*  *min*)

(2)

noted earlier how the data changed when going from a slower to faster pace. There is less apparent variation due to the spring with a longer, quicker stride. The result was a higher average for the sample points, with respect to the data minima and maxima.

Visually, it is a little difficult to be sure of this, given the amount of bounce in the steps shown in [Figure 4](#_bookmark3). But calculations showed that the average-vs.-peak values are very similar to those in [Figure 2](#_bookmark2). Therefore, a possible simple algorithm was used to determine the distance walked. It is

(*avg*  *min*)

This algorithm held up well for a variety of subjects and paces, with all variation within approximately 6%. The algorithm lends itself to easy calibration for a specific individual/pace by adjusting the Multiplier k. There is also provision within the listed code to perform an average on the stride length to smooth out step- to-step variation. The results noted here did not include the use of this averaging.

In this experiment, only the X- and Y-axes were used. The

3-axis accelerometer was chosen for flexibility, and two axes were found to be adequate for the task. An ADXL323 could

*d*  *k*  (*max*  *min*)

where:

*d* is the distance calculated.

*k* is a constant multiplier.

(1)

be used in place of the ADXL330. The same layout can be used for both because the pin configuration is identical except for the Z-axis output.

These experiments concentrated on achieving good results for the pedometer’s distance measurement. There was no extensive

*max* is the maximum acceleration value measured within this step.

*min* is the minimum acceleration value.

*avg* is the average acceleration value for the step.

Equation 1 is completed for each step, as determined by a different step-finding algorithm. The step-finding algorithm uses an 8-point moving average to smooth the data. It searches for a maximum peak, followed by a minimum. A step is counted when the moving average crosses the zero point, which is the overall average for the step. The data used in the distance algo- rithm takes into account the 4-point latency of the moving average.

This simple solution held up well for Subject 1 over various stride lengths. It also did reasonably well with additional subjects. But some subjects produced distances that varied as much as 10% from the average measured distance for the group. This was not within the ±7.5% error band that was targeted for an uncalibrated measurement. Another solution was needed.

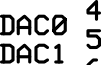
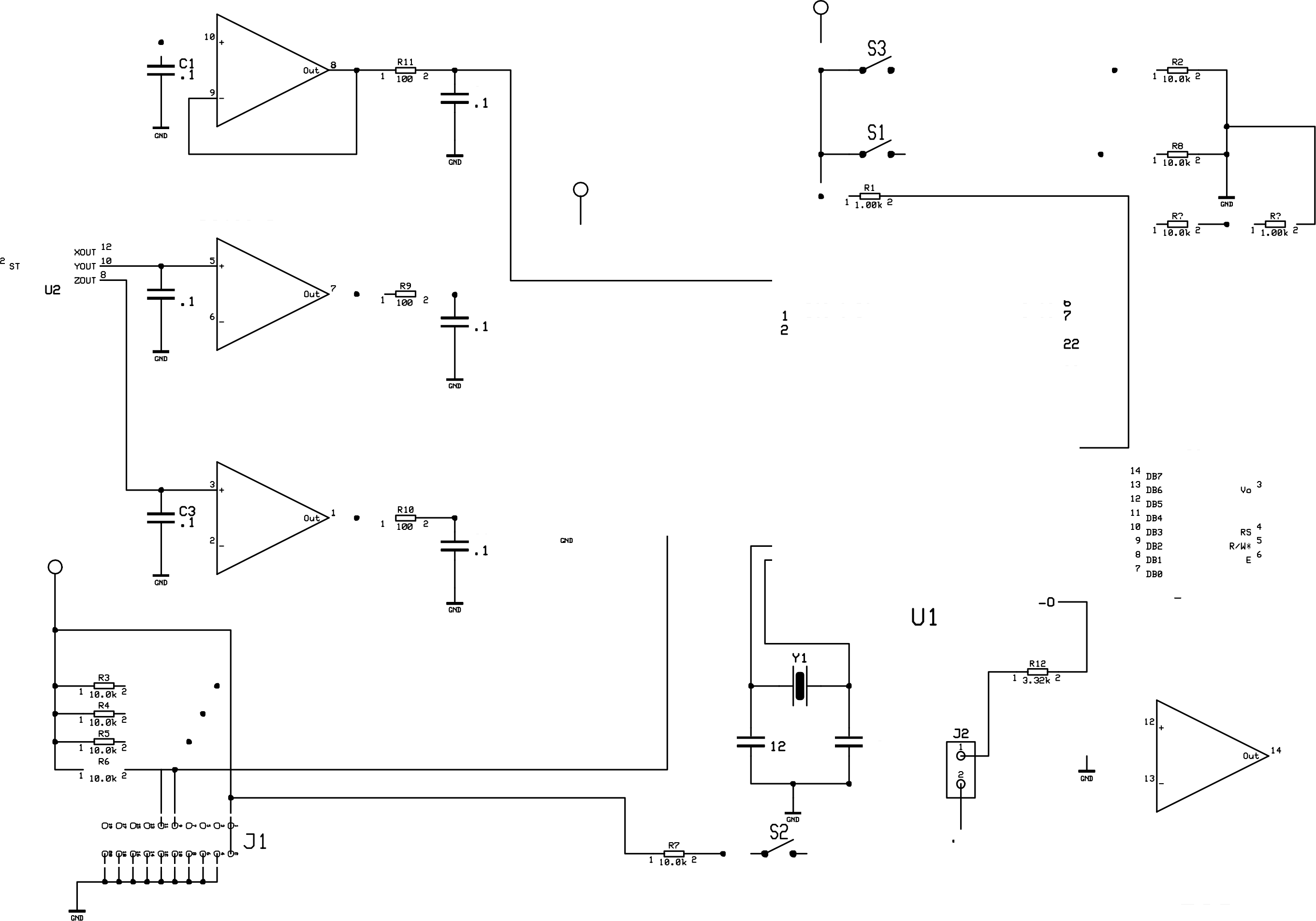
work done on the step-counting algorithm beyond ensuring it works well while walking (and running), with measured steps within 1 to 2 steps of the actual number over hundreds of steps taken. However, it is possible that a simple algorithm can be fooled with nonwalking motions. Improvements can be made in that area, such as the time-window function described in AN-602. The object is to ignore steps that are outside the expected time window, while retaining the ability to adapt when the subject changes pace.

#### SUMMARY

This application note represents the results of a single set of experiments attempting to gain decent performance from a simple pedometer that uses a single accelerometer. Some of the barriers to gaining that performance have been reviewed.

The final results have met the stated accuracy goals, with the added possibility of improved accuracy with calibration. Although greater accuracy can be obtained with a more complex system (for instance, with multiple accelerometers), the algorithm provided in this application note is an excellent starting point for simple, low cost applications.

##### AD8609ARU



C4

##### AD8609ARU

ADuC7020BCP

ADXL33eKCP

C5

ADC4

Di3C3

PE. 8ZSPN9xPLAO

AD8609ARU

•••L CI4

\*\* 16

18

. 47

PB. 3xTRST

PB. 4ZINT6ZPLAO

PB. 6<NRST PLA0

PB. 5 INT1ZPLAO

27

28

*EB*

21

PI. 1<SPN1SPLA 1

PI. 2 SPN2 PLR1 P1. 3 SPN3 PLCI P1. 4 SPN4xPLFl 1 P1. 57SPN57PLA I

PI. 6<SPN6 PLR I PI 7<SPN7 PLA0

’\*

XCLKI

XCLKO

VREF LVDO

TIIS

3Z

31

O0

2

DVDD3. 3

C6 —

26

32.7682Hz

15

0D8609DRU

DR 1

**AN-900**

*Figure 5. Custom Board Schematic*

Rev. 0 | Page 4 of 16

LED p1ugs @ i n here

TOP

0

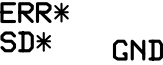
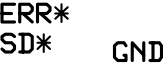
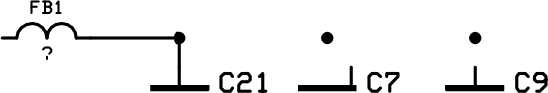
0

7

6

061 - 5

ADP3330ART-5



VR2

OUT

I N

2

## 3

. 47 6

1

C2B C17 CB

1euF . 47 . 1

**AN-900**

Rev. 0 | Page 5 of 16

### ” ADP3330ART—3.3

*Figure 6. Custom Board Schematic (Continued)*

VR1

2 IN " OUT

## 3

6

##### DVDD3.3

1BuF •••••j . 1 . 1

# Pouer

06617-006

**ADuC7020 C CODE**

The following three files contain the code used to implement this algorithm on the test boards using an ADuC7020 microcontroller.

#### MAIN.C

The file *main.c* is used to initialize the device:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Author : J Scarlett

Date : Nov 2006

Files : main.c, display.c, ped.c

Hardware : ADuC7020

Description : Implements a simple pedometer based on application note AN-602

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ #include <ioaduc7020.h>

// Function Prototype with required function attribute. extern void Monitor\_Function(void);

extern void Display\_Init(void);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Main Function for ADuC7020 Pedometer essentially performs startup functions

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* int main (void)

{

POWKEY1 = 0x01;

POWCON = 0x06; // set to 653kHz core clock

POWKEY2 = 0xF4;

REFCON = 0x2; // use external reference

// (connected to VDD)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Initialize Peripherals

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// GPIO Configuration

GP4DAT = 0x04000000; // P4.2 configured as an output.

// LED is turned on

GP0CON = 0x00000000; //

GP0DAT = 0xE0000000; // 0.7, 0.6, and 0.5 are outputs

// 0.7 = E, 0.5 = R/W\*, 0.6 = RS

GP1DAT = 0xFF000000; // All P1 pins are outputs

ADCCON = 0x20; // Turn ADC on but do not enable

Display\_Init(); // found in File display.c

Monitor\_Function(); // found in File ped.c

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Main Loop

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* while(1)

{

}

} // main()

#### DISPLAY.C

The file *display.c* contains all functions that access the display:

// File "display.c"

// performs all LCD display interface functions #include <ioaduc7020.h>

extern char stepbcd[6]; // found in File ped.c

extern char distbcd[6]; // this too

void Display\_Init(void); void display\_data(void);

void display\_data\_clear(void); void char\_load(int RS, int data); void delay(unsigned int cycles); int reverse\_data(int data);

void Display\_Init()

{

int rs, data;

// used once to initialize display and write

// the "Steps" and "Distance" headers

// Display initialization

rs = 0; // no RAM access yet

data = 0x30; // function set: 2-line mode, display off char\_load(rs, data);

data = 0x38; // function set: 2-line mode, display off char\_load(rs, data);

data = 0x08; // display off, cursor off, blink off char\_load(rs, data);

data = 0x01; // clear display

char\_load(rs, data);

delay(49); // ~1.5 ms additional delay is required

data = 0x06; // increment mode, no shift char\_load(rs, data);

data = 0x0C; // display on, cursor off, blink off char\_load(rs, data);

data = 0x80; // set data address to home, just to be sure char\_load(rs, data);

rs = 1; // now writing to data RAM

data = 0x53; // start of sequence to send

char\_load(rs, data); // "Steps Distance" for title line

data = 0x74; // "t"

|  |  |  |  |
| --- | --- | --- | --- |
| char\_load(rs, | data); |  | |
| data = 0x65; char\_load(rs, | data); | // | "e" |
| data = 0x70; char\_load(rs, | data); | // | "p" |
| data = 0x73; char\_load(rs, | data); | // | "s" |
| data = 0x20; char\_load(rs, | data); | // | " " times 3 |
| char\_load(rs, char\_load(rs, | data);  data); |  |  |
| data = 0x44; char\_load(rs, | data); | // | "D" |
| data = 0x69; char\_load(rs, | data); | // | "i" |
| data = 0x73; char\_load(rs, | data); | // | "s" |
| data = 0x74; char\_load(rs, | data); | // | "t" |
| data = 0x61; char\_load(rs, | data); | // | "a" |
| data = 0x6E; char\_load(rs, | data); | // | "n" |
| data = 0x63; char\_load(rs, | data); | // | "c" |
| data = 0x65; char\_load(rs, | data); | // | "e" |

} // Display\_Init()

void display\_data()

{

int i, rs, data, zero;

// displays the data contained in stepbcd[] & distbcd[]

// beginning at the first and ninth characters

// on row 2 of the display

rs = 0; // want to set address, not data

data = 0xC0; // start of second line

char\_load(rs, data);

rs = 1;

zero = 0;

for (i=5; i>=0; i--) // display steps

{

if ((stepbcd[i] > 0) || (zero ==1)) // suppress leading zeroes,

{ // but not embedded zeroes

zero = 1;

data = 0x30 | stepbcd[i]; // numbers on display character table

// begin at 0x30

} // for

} // if

char\_load (rs, data);

rs = 0; // set address

data = 0xC8; // ninth character of second line char\_load(rs, data);

rs = 1;

zero = 0;

for (i=5; i>=0; i--) // display distance

{

if ((distbcd[i] > 0) || (zero ==1))

{

} // for

} // if

zero = 1;

data = 0x30 | distbcd[i]; char\_load (rs, data);

} // display\_data()

void display\_data\_clear(void)

{

int i, rs, data;

// used to clear display data field

// before new measurement

rs = 0; // want to set address

data = 0xC0; // start of second line

char\_load(rs, data);

rs = 1; data =0x20;

for (i=0; i<16; i++) // put spaces across Row 2 char\_load (rs, data);

} // display\_data\_clear()

void char\_load(int rs, int data)

{

// signal timing assumes a core clock < 4MHz

// delay at end is ~61us, to allow write to complete

data = reverse\_data(data); // board layout dictated reversing MSB/LSB

GP1CLR = 0x00FF0000; // ensure clean slate for next character

GP1SET = data << 16; // set Port 1 to new character data

if (rs) else

GP0SET = 0x00400000; // RS bit = 1

GP0CLR = 0x00400000; // RS bit = 0

GP0CLR = 0x00200000; // WR bit = 0 (this is a write command)

GP0SET = 0x00800000; // set E bit to begin transfer process

GP0CLR = 0x00800000; // clear E bit to complete transfer process delay(2);

} // char\_load()

void delay(unsigned int cycles)

{

T2CON = 0;

T2CLRI = 0;

T2LD = cycles;

T2CON = 0xC0; // enable Timer 2, periodic mode, 32.768 kHz

while (!(IRQSIG & WAKEUP\_TIMER\_BIT)); // wait for timeout

T2CON = 0; // disable Timer 2

} // delay()

int reverse\_data(int data)

{

int i, temp; temp = 0;

for (i=0; i<4; i++)

{

temp |= (((0x01 << i) & data) << (7 - (2 \* i)));

} // fill top 4 bits of temp

for (i=4; i<8; i++)

{

temp |= (((0x01 << i) & data) >> ((2 \* i) - 7));

} // fill bottom 4 bits of temp return temp;

} // reverse\_data()

#### PED.C

The file *ped.c* contains those functions used by the pedometer algorithm:

// file "ped.c"

// performs pedometer and misc functions

#include <ioaduc7020.h> #include <math.h> #include <stdlib.h>

// Function Prototype & variables char stepbcd[6];

char distbcd[6]; char stepflag;

float stride, avgstride, accel\_dat[50];

float maxavg, minavg, accel\_avg, velocity, displace; float distance;

int steps;

void Monitor\_Function(void);

void get\_sample(unsigned int \*xdat, unsigned int \*ydat, unsigned int \*zdat); char IsStep(float avg, float oldavg);

void display\_prep(void);

long int bin\_to\_bcd(long int bin\_no);

extern void display\_data(void); // found in File display.c

extern void display\_data\_clear(void); // found in File display.c

// functions

void Monitor\_Function()

{

char flag;

unsigned int xdat, ydat, zdat;

int i, cycle\_count, tot\_samples, avgconst = 1, latency = 4, avglen = 8; float rssdat, newmax, newmin, oldavg, newavg, avgthresh=1.0;

float walkfudge = 0.0249;

flag = 0;

T1CON = 0; // turn off interval timer and clear any IRQ T1CLRI = 0;

while (1)

{

if (IRQSIG & XIRQ0\_BIT) // XIRQ0 button has been pressed

{

while(GP0DAT & 0x00010); // wait for XIRQ to be low again if (!flag)

{

T1CON = 0; // turn off interval timer

T1CLRI = 0; // clear any timer IRQ

stepflag = 2;

maxavg = -10000.0;

minavg = 10000.0;

newmax = -10000.0;

newmin = 10000.0;

oldavg = 0.0;

newavg = 0.0;

cycle\_count = 0;

tot\_samples = 0;

steps = 0;

distance = 0.0;

accel\_avg = 0.0;

velocity = 0.0;

displace = 0.0;

avgstride = 0.0;

display\_data\_clear(); // clear old data from display flag = 1;

T1LD = 1092; // ~30 Hz sample rate

T1CON = 0x2C0; // 32.768 kHz clock, timer on,

// periodic mode

} // if not running, start.

} // look for stop button if (GP2DAT & 0x01)

{

while(GP2DAT & 0x01); flag = 0;

} // if running, stop

if (((IRQSIG & GP\_TIMER\_BIT) && (flag)) != 0) // wait for timeout

// and flag

{

T1CLRI = 0;

if (tot\_samples > 7) // subtract first sample in sliding boxcar avg

{

oldavg = newavg;

newavg -= accel\_dat[cycle\_count - avglen];

} // if

get\_sample(&xdat, &ydat, &zdat); // get data from accelerometer xdat -= 8192; // subtract Zero g value

ydat -= 8192;

rssdat = sqrt((float)(xdat\*xdat + ydat\*ydat)/16.0); // vector sum accel\_dat[cycle\_count] = rssdat; // place current sample data in buffer

newavg += rssdat; // add new sample to sliding boxcar avg if((abs(newavg-oldavg)) < avgthresh)

newavg = oldavg;

if (rssdat > newmax) newmax = rssdat;

if (rssdat < newmin)

newmin = rssdat;

tot\_samples++;

cycle\_count++; // increment count of samples in current step

if (tot\_samples > 8)

{

if (IsStep(newavg, oldavg))

{

for (i = latency; i < (cycle\_count - latency); i++) accel\_avg += accel\_dat[i];

accel\_avg /= (cycle\_count - avglen);

for (i = latency; i < (cycle\_count - latency); i++)

{

velocity += (accel\_dat[i] - accel\_avg); displace += velocity;

} // create integration and double integration

// calculate stride length

stride = displace \* (newmax - newmin) / (accel\_avg - newmin); stride = sqrt(abs(stride));

// use appropriate constant to get stride length stride \*= walkfudge;

// generate exponential average of stride length to smooth data if (steps < 2)

avgstride = stride;

else

avgstride = ((avgconst-1)\*avgstride + stride)/avgconst;

steps++;

distance += avgstride;

// need all data used in calculating newavg for (i = 0; i < avglen; i++)

accel\_dat[i] = accel\_dat[cycle\_count + i - avglen];

cycle\_count = avglen; newmax = -10000.0;

newmin = 10000.0;

maxavg = -10000.0;

minavg = 10000.0;

accel\_avg = 0;

velocity = 0;

displace = 0;

display\_prep(); display\_data();

// temporary

if (GP4DAT & 0x04) // toggle LED to reflect step GP4CLR = 0x040000;

else

GP4SET = 0x040000;

} // we have a new step

} // enough samples to start checking for step (need at least 8)

} // if timeout

} // continual loop

} // Monitor\_Function()

void get\_sample(unsigned int \*xdat, unsigned int \*ydat, unsigned int \*zdat)

{

int i;

\*xdat = 0;

\*ydat = 0;

\*zdat = 0;

for (i=0; i<15; i++)

{

// gets new samples for x, y, z axes

// sums together 4 measurments to get average

ADCCP = 0; // x axis

i++; // delay one command cycle

ADCCON = 0xA3; while (!(ADCSTA));

\*xdat += ((ADCDAT >> 16) & 0xFFF); // data is in bits 16 - 27, so shift is necessary

ADCCP = 1; // y axis

i++;

ADCCON = 0xA3; while (!(ADCSTA));

\*ydat += ((ADCDAT >> 16) & 0xFFF);

} // for

ADCCP = 2; // z axis

i++;

ADCCON = 0xA3; while (!(ADCSTA));

\*zdat += ((ADCDAT >> 16) & 0xFFF);

} // get\_sample()

char IsStep(float avg, float oldavg)

{

// this function attempts to determine when a step is complete

float step\_thresh = 5.0; // used to prevent noise from fooling the algorithm

if (stepflag == 2)

{

if (avg > (oldavg + step\_thresh)) stepflag = 1;

if (avg < (oldavg - step\_thresh)) stepflag = 0;

return 0;

} // first time through this function

if (stepflag == 1)

{

if ((maxavg > minavg) && (avg >

((maxavg+minavg)/2)) && (oldavg < ((maxavg+minavg/2))))

return 1;

if (avg < (oldavg - step\_thresh))

{

stepflag = 0;

if (oldavg > maxavg)

maxavg = oldavg;

} // slope has turned down return 0;

} // slope has been up

if (stepflag == 0)

{

if (avg > (oldavg + step\_thresh))

{

stepflag = 1;

if (oldavg < minavg)

minavg = oldavg;

} // slope has turned up return 0;

} // slope has been down return 0;

} // IsStep()

void display\_prep()

{

int i;

long int temp;

// convert steps to BCD values for sending to display temp = steps;

temp = bin\_to\_bcd(temp); // function to convert binary

for (i=0; i<6; i++) // to BCD

{

} // for

stepbcd[i] = (char)(0xF & temp); // load each digit temp = temp >> 4;

// convert distance to BCD values for sending to display

temp = (long int)(distance); // convert float to long int temp = bin\_to\_bcd(temp);

for (i=0; i<6; i++)

{

} // for

distbcd[i] = (char)(0xF & temp); // load each digit temp = temp >> 4;

} // display\_prep()

long int bin\_to\_bcd(long int bin\_no)

{

int i;

long int divisor, multiplier, bcd\_no, temp;

divisor = 100000;

multiplier = 1048576;

bcd\_no = 0;

temp = 0;

if (bin\_no > 999999)

bin\_no = 999999;

for (i=0; i<6; i++)

{ // determine each digit starting

temp = bin\_no/divisor; // with most significant

bin\_no -= temp\*divisor; // subtract this amt

temp \*= multiplier; // generate hex equivalent

bcd\_no += temp; // put bcd value together

} // for

divisor /= 10; // go to next digit

multiplier = multiplier >> 4;

return bcd\_no;

} // bin\_to\_bcd()

**©2007 Analog Devices, Inc. All rights reserved. Trademarks and registered trademarks are the property of their respective owners.**



**AN06617-0-7/07(0)**