Muggling!

A Wireless Spherical Multi-Axis Musical Controller

Pascal Stang John R. McCarty Jeffrey Traer Bernstein
Music 250a, Stanford University, Fall 2002

1 Introduction

Traditional musical instruments require formal training and an understanding of musical theory to play. Most electronic or computer based musical instruments are either fettered by restrictive slider knobs and buttons or insufficiently constrained to be used effectively (as is the case with the instruments such as the Theremin). Humans have a learned reaction to spherical objects due to tradition interaction with objects of this shape. The typical spherical object encountered is meant to be thrown, bounced, hit, or squeezed, so we all of a basic idea of what can be done with a ball. A particularly visually spectacular use of balls is juggling, and due to its theme and variation nature provides excellent input of sonification, thus is born music + juggling = Muggling!

1.1 Prior Work

Prior work in juggling as an interface for computer music has focused on the gestures of the performers and not on data acquired from juggling objects. In [3], sensors were placed on the jugglers wrists and the only communication with the juggling pins was to control their illumination. Prior work on balls as a musical interface has centered on squeezing of balls as in [4]. Sensing of the movement of an object for musical expression has mostly been used for conducting (or simulation of conducting) of orchestras as in [1],[2], or Max Mathews' Radio Batton.

1.2 System

The Muggling! ball contains four two-axis accelerometers, four RGB LEDs a microprocessor and a transmitter receiver pair. Data regarding the ball's acceleration is transmitted to the base which contains a microprocessor and transmitter receiver pair, that converts the data to midi signals which are routed to a computer which provides the musical mapping.

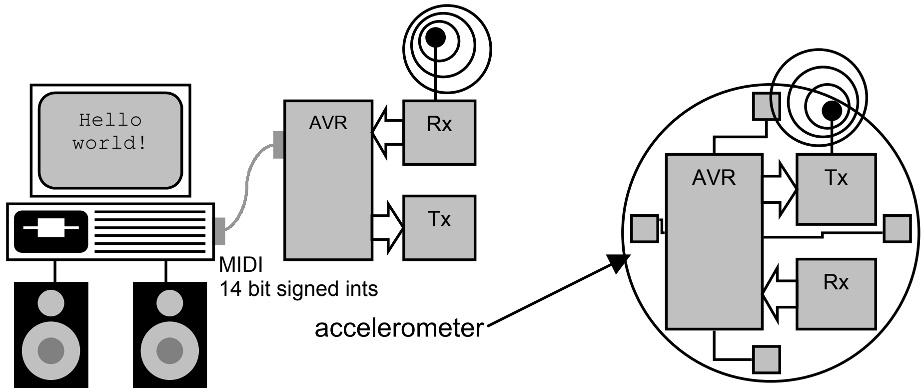


figure 1*, Receiver and Ball Transmitter Block Diagram*

2 Musical Mappings

2.1 Chinese Baoding Ball

This mapping is similar to Chinese Baoding balls and has a similar relaxing quality. While Baoding balls have a single plate inside for producing sound, our electronic version has 12 biquad resonators that are triggered with an impulse based on the amount of acceleration in X and Y directions. Figure 2 provides a conceptual view of the instrument. Chimes are spread around the perimeter of the ball and an simulated actuator rolls around inside the ball as it is rotated and strikes the chimes. See appendices for pd implementation.

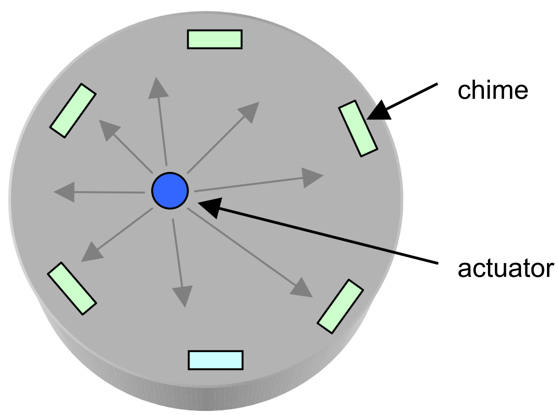


***figure 2**, model of chimes in ball*

2.2 Scratching

In scratching the tilt in one dimension is mapped to a position within an audio file so that rotating or pushing the ball back and forward is equivalent to moving a record, producing the "scratch" sound popular in Hip Hop music and its derivatives. Shaking the ball in one dimension as actually fairly analogous to the actual motion. Using the ball overcomes the physical limitations of the medium such as skipping, and requires less skill since many motions will produce the desired effect instead of one technique that requires significant practice to master. In fact a wide range of motions make for interesting sounds including contact juggling, which proved to be visually intriguing as well since the juggler's movements abstracted the connection between specific movement and sound but still demonstrated a causal connection.

In scratching the tempo of the background beat was set by shaking the ball. The running average of the time between 4 shakes was taken and the playback speed of the background beat is set.

2.3 Two-Dimensional Visualization and Four Channel Panning

A particularly educational mapping is tilt in two dimensions to two-dimensional position. When implemented in pd, this allowed us better understand the input we were receiving from the ball. Mapping this input to four channel panning the user can use the ball to place sound two-dimensionally within a room

3 Implementation

3.1 Ball Transmitter

The ball contains 4 Analog Devices 2-axis 2G accelerometers, two are mounted facing up and two are mounted facing the exterior of the ball. The accelerometers provide about 8 bits of usable resolution which sampled at 100 Hz by the analog to digital converters on the Atmel microprocessor. The data from the sensors is then filtered with a one-pole lowpass filter to prevent aliasing since the transmission occurs at a slower rate, the filtering also interpolates the data to a further signed 14 bits which are transmitted to the base via the Linx Transmitter (a receiver is also included to allow for timing of communications between multiple balls in the future). The ball is powered by 4 1.3 V NiMH batteries mounted to ensure gravity is properly centered.

	a_{linear}	$a_{angular}$
x	$(x_1+x_2)/2$	$(x_1-x_2)/2$
y	$(y_1-y_2)/2$	$(y_1+y_2)/2$
z	$(z_1-z_2)/2$	$(z_1+z_2)/2$

***table 1**, sensor calculations*

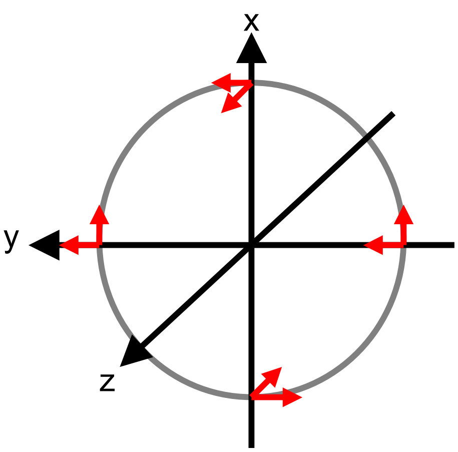


***figure 3**, sensor geometry*

Linear and angular acceleration are calculated from the two-dimensional accelerometer data by adding or subtracting the appropriate dimensions from each sensor as displayed in figure 3 and table 1.

Four RGB LEDs where also added to the ball these changed through 16 discrete values of each colour according to the data from each sensor.

3.2 Receiver Base

The receiver base receives data from the ball sensors and converts it to MIDI to be sent to an instrument (in this case a linux computer running PD). The 14 bit signed integers received are encoded as two 7 bit numbers sent as note and velocity over MIDI, these values are then decoded in PD before they are used. The base consists of Atmel microprocessor, a Linx receiver-transmitter pair, and a liquid crystal display. Instant feedback from the sensors is displayed on the LCD.

4 Conclusions and the Future

Our design from inception to completion remained fairly consistent, and the final product was relatively close to the initial concept. Thanks to incredible hardware design knowledge the most crucial design element, wireless capability, was achieved successfully. The biggest design change was not making the ball able to bounce, which was mostly a product of time limitations. At a late stage we decided not to include pressure sensors on the surface of the ball. This decision was made partly for aesthetics and also having pressure sensors of the hard shell of the ball affected its contour and ability to roll. Also, if the device were used for juggling the pressure sensors would not be useful since very little squeezing would occur. Our musical concept for the ball was as a juggling instrument, either as a percussive ground juggling device or more melodic traditional juggling controller. However, since the ball ended up being rather fragile and not bouncy this concept was pushed aside. This is probably the most unsatisfying aspect of the final product. Having a completely wireless spherical controller offers so many possibilities that we were unable to adequately explore without potentially destroying the device. In the end our ball could have had wires attached and still controlled our demonstration patches perfectly. Our example mappings and patches did not appropriately exploit the wireless quality of the device.

There are several future enhancements that could be made to improve the usability and function of the device. Obviously it needs to bounce. Also, the addition of 2 more accelerometers would provide complete measurement of motion in each plane. It would be ideal to have several balls that can be used simultaneously; this presents several hurdles. If the balls all transmit on the same frequency then their transmissions must be shared. It might be possible to have a separate transmitter receiver for each ball, but then the hardware requirements and costs would grow rapidly. There are several problems that must be solved before a complete set of "muggling" balls can be produced. Another interesting enhancement for use as a handheld device is the addition of 2 motor driven perpendicular gyroscopes inside the ball that would provide haptic feedback through resistance to tilt.

Ultimately, we were satisfied with our final product. It lived up to most of our design expectations. The ball was both comfortable to hold and pleasant to look at, and the music results were both interesting and entertaining.

References

[1] Teresa Marrin and Joseph Paradiso, "The Digital Baton: a Versatile Performance Instrument" *Proceedings of the International Computer Music Conference*, pp. 313-316, Thessaloniki, Greece, September 1997.

[2] Jan Borcher, "WorldBeat: Designing a Baton-Based Interface for an Interactive Music Exhibit", *Proceedings of ACM CHI '97*, Atlanta, Georgia, March 1997.

[3] Mathew Reynolds, et al., "An Immersive, Multi-User Musical Stage Environment", *Proceedings ACM Siggraph 2001*, ACM Press, NY.

[4] Weinberg, G., Orth M., and Russo P. (2000) "The Embroidered Musical Ball: A Squeezable Instrument for Expressive Performance," *Proceedings of CHI 2000*. The Hague: ACM Press.

A PD Patches

Provided with acceleration and angle in three axes via midi from the ball, the ballout patch was used to decode the data, calculate the total magnitude of acceleration, and pipe it to one of seven outlets.

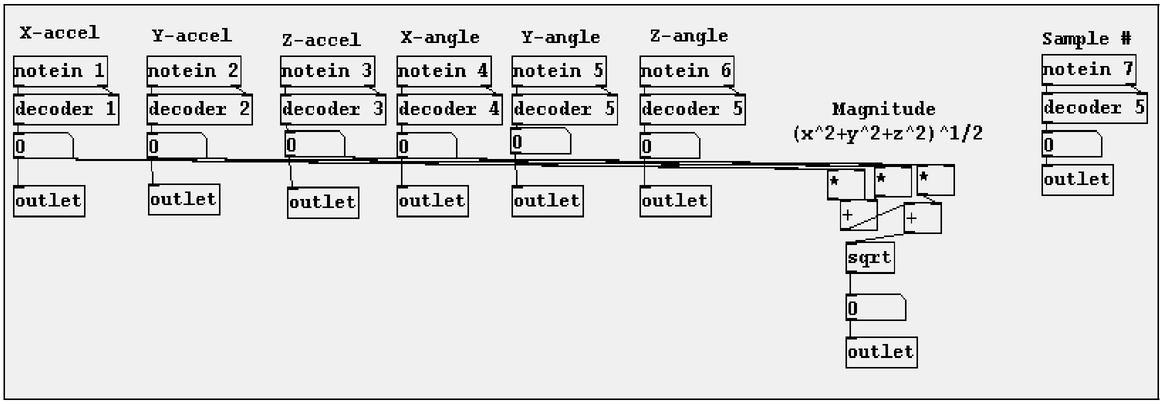


Figure 1 - ballout.pd

The decoder patch is used to translate the 14 bit 2's complement data sent from the AVR as midi note and velocity values into floating point values for use in pd.

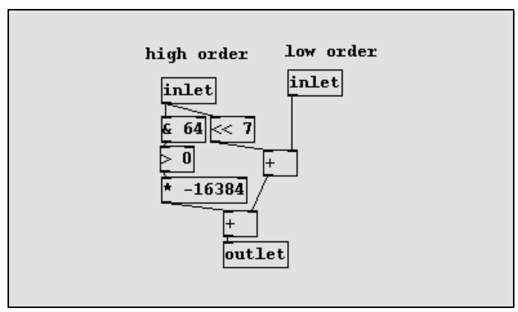


Figure 2 - decoder.pd

The chime patch was implemented using the biquad~ object in pd. By setting the coefficients appropriately a constant gain resonator with sweepable center frequency is created. When this filter is sent a bang as input, it rings like a chime. The full patch is shown at the end of this section which is made up of 12 resonators set to different frequencies. The X and Y accelerations are each mapped to six resonators, as the ball is accelerated in either X or Y direction the chimes are banged, with each chime mapped to a specific acceleration value.

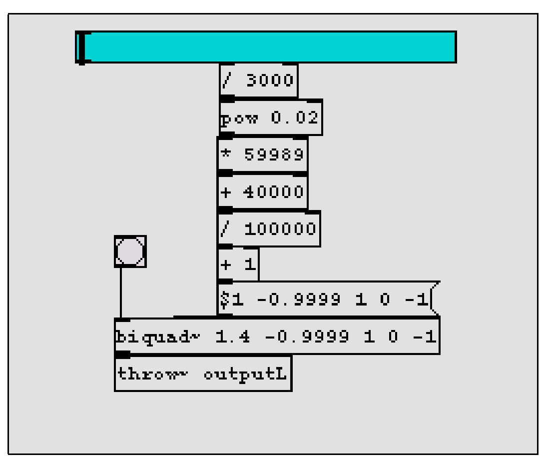


Figure 3 - resonator

When using the ball as a hand held controller, tilt values are mapped to signal controls. The graphic grid object gives visual feedback for the balls tilt position in the X-Y plane. Below, the positional display is combined with a 4-channel panning patch. The ball can be used to move sound around in space while not confining the user to a particular location within the room.

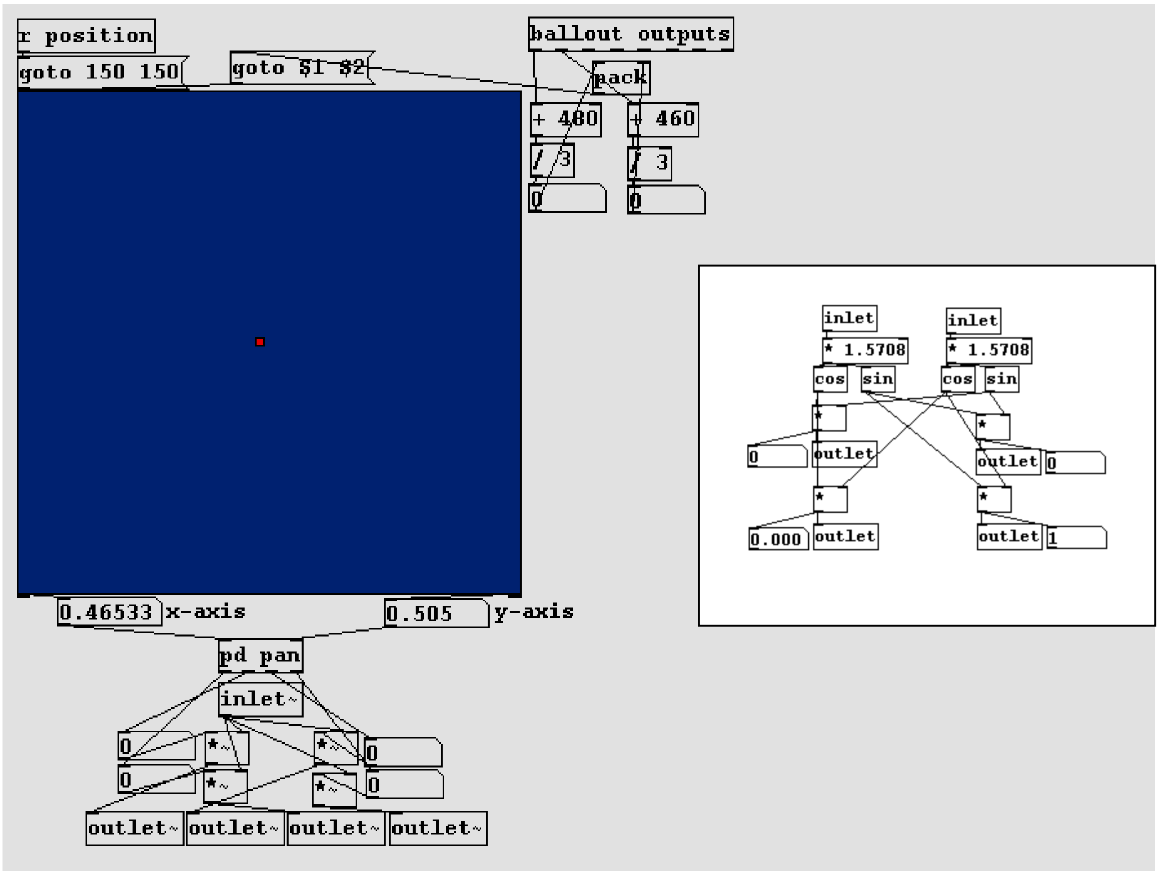


Figure 4 - Four channel panner

The scratch patch maps X-tilt value to a position within an audio file. Tilting the ball left and right scrolls through the file at a rate proportional to the velocity of ball movement. The orange horizontal slider gives graphic feedback for the current scratch position within the sample file. Although simple, this seemed to be a hit with the audience.

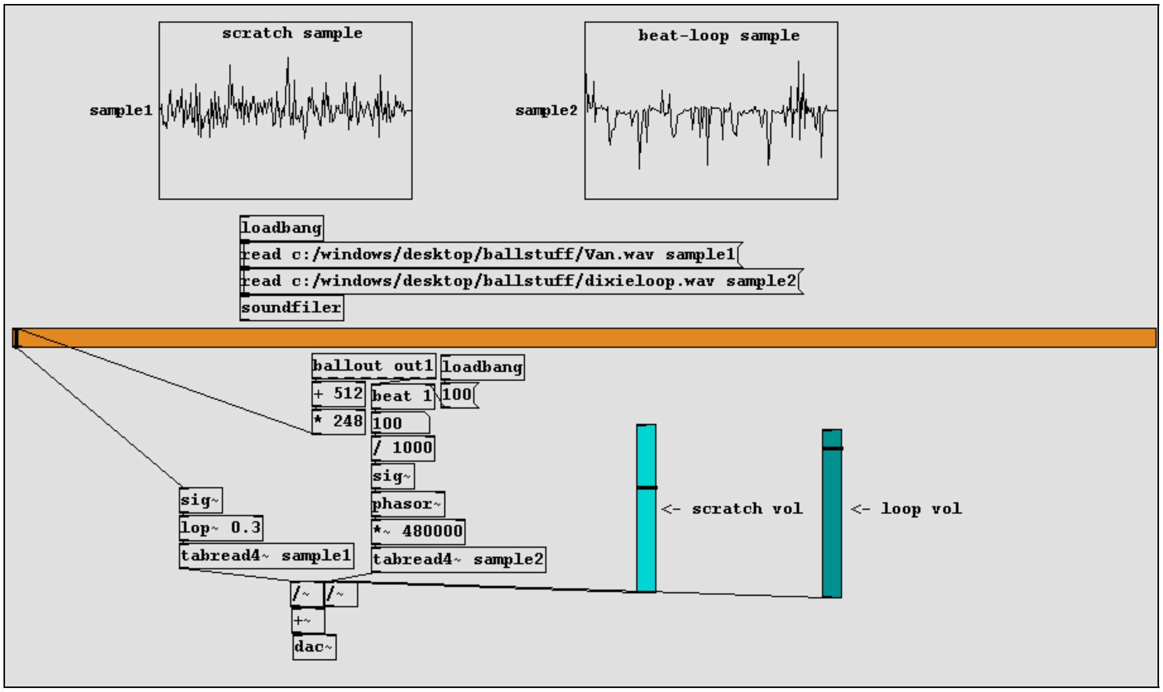


Figure 5 – scratcher.pd

The beat external object used in scratcher.pd is mapped to magnitude of acceleration. When the ball is accelerated with a large amount of force the magnitude surpasses a preset threshold and sends out a bang.
A timer is used to measure the time between successive bangs, which is converted to a bpm value. An average of four successive measurements is taken and a temp value is sent from the outlet. This allows the tempo to be adjusted by shaking the ball in time at the desired tempo. The threshold can be adjusted so that the tempo is not affected unless a large amount of force is applied which avoids accidental tempo changes. A slight modification of this patch would allow the ball to be used as an electronic shaker or maraca type instrument. Also this can be used for playback control of a midi score or preset sequence of notes.

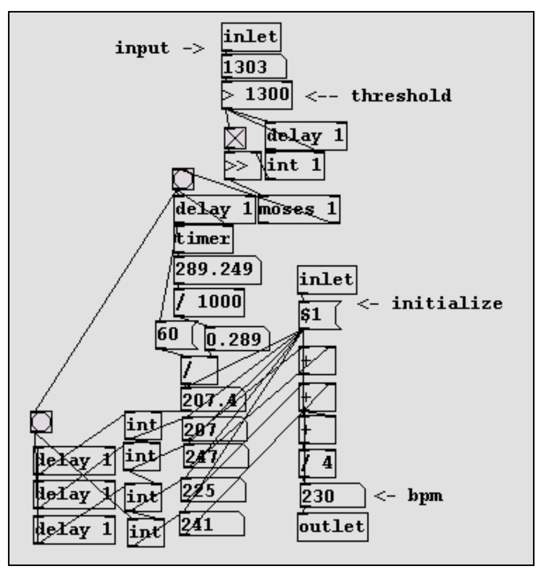


Figure 6 - beat.pd

Below are two abandoned patches that mapped ball movement directly to audio parameters. Since the acceleration values change very rapidly and almost arbitrarily as the ball rolls, output of these patches was not very satisfying was quite random. The simple patch maps acceleration to oscillator frequency. While the buzz patch has predefined pitches that are mapped to particular ranges of the X-acceleration, Y and Z acceleration are mapped to timbre and harmonic content, and magnitude is mapped to volume. The buzz patch may have been more successful if ball could bounce or was less fragile so it could be safely tossed into the air.

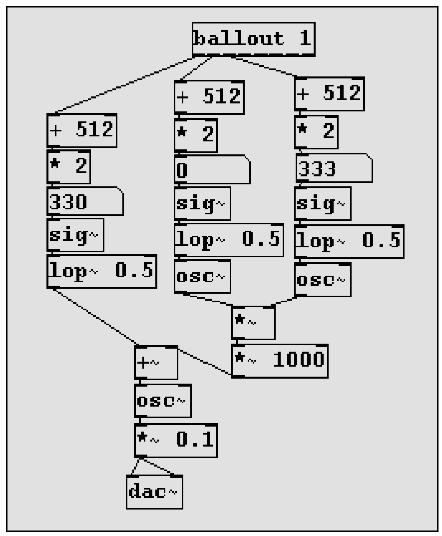


Figure 7 - simple.pd

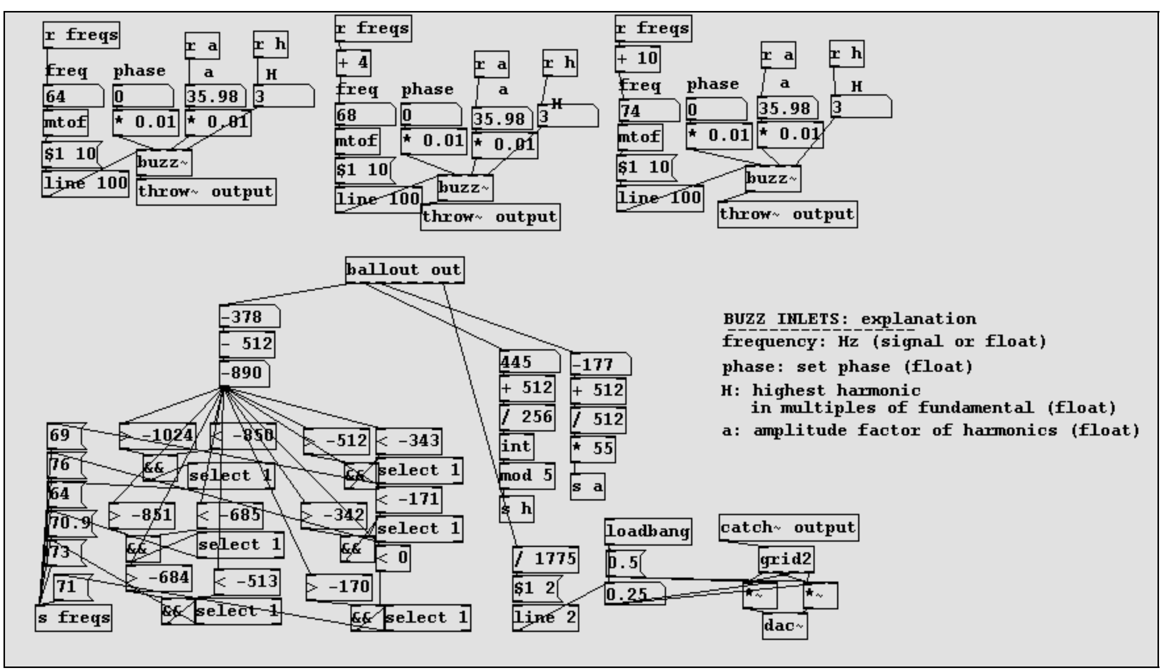


Figure 8 - buzz.pd

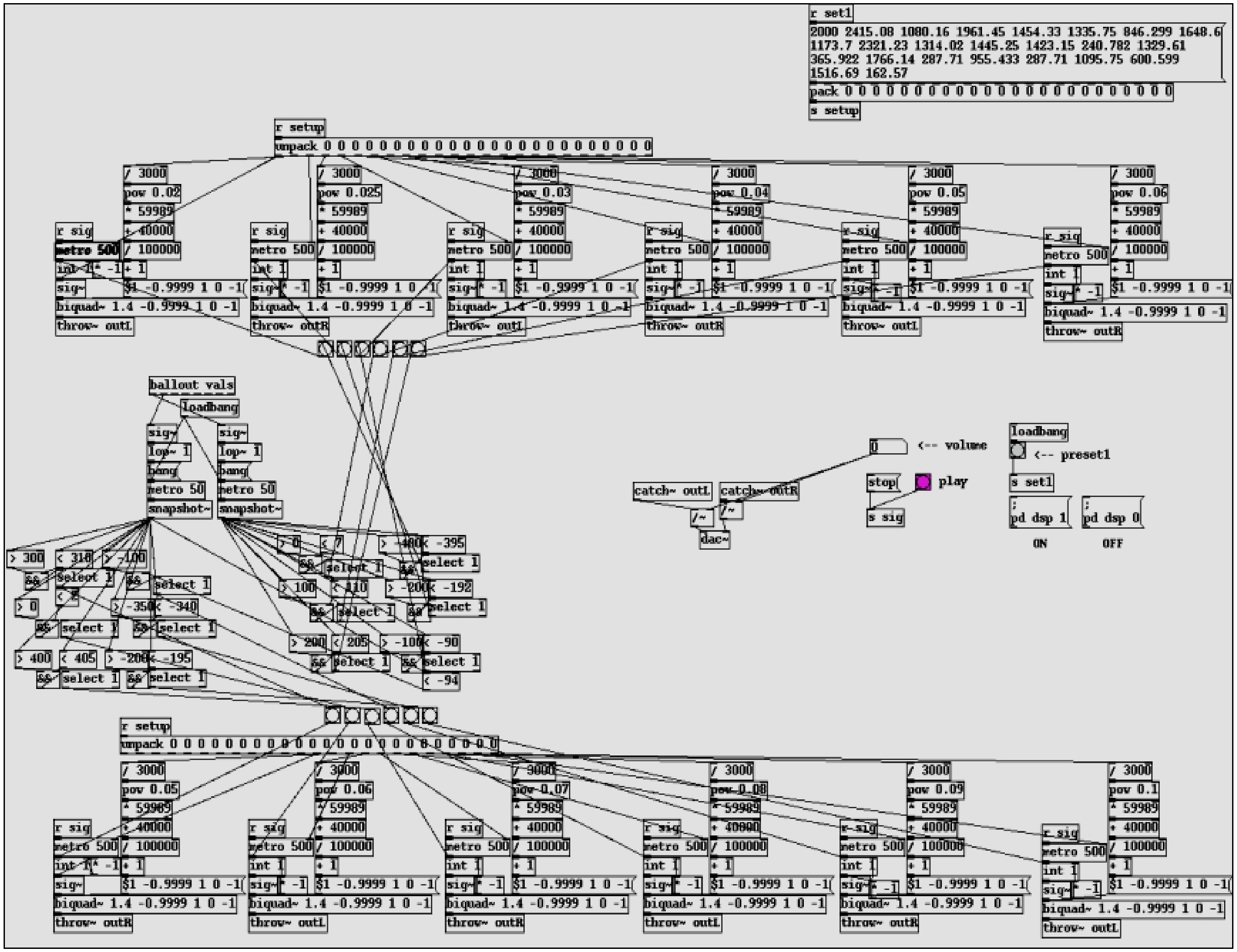
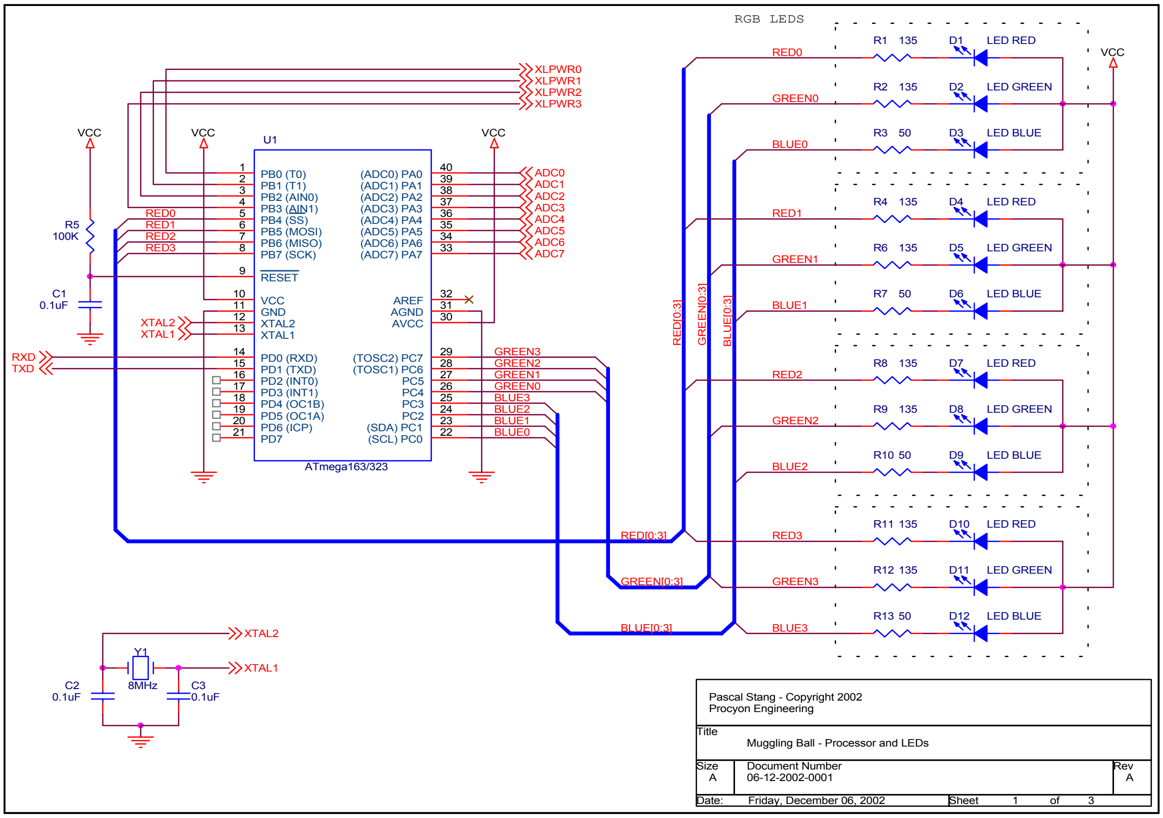
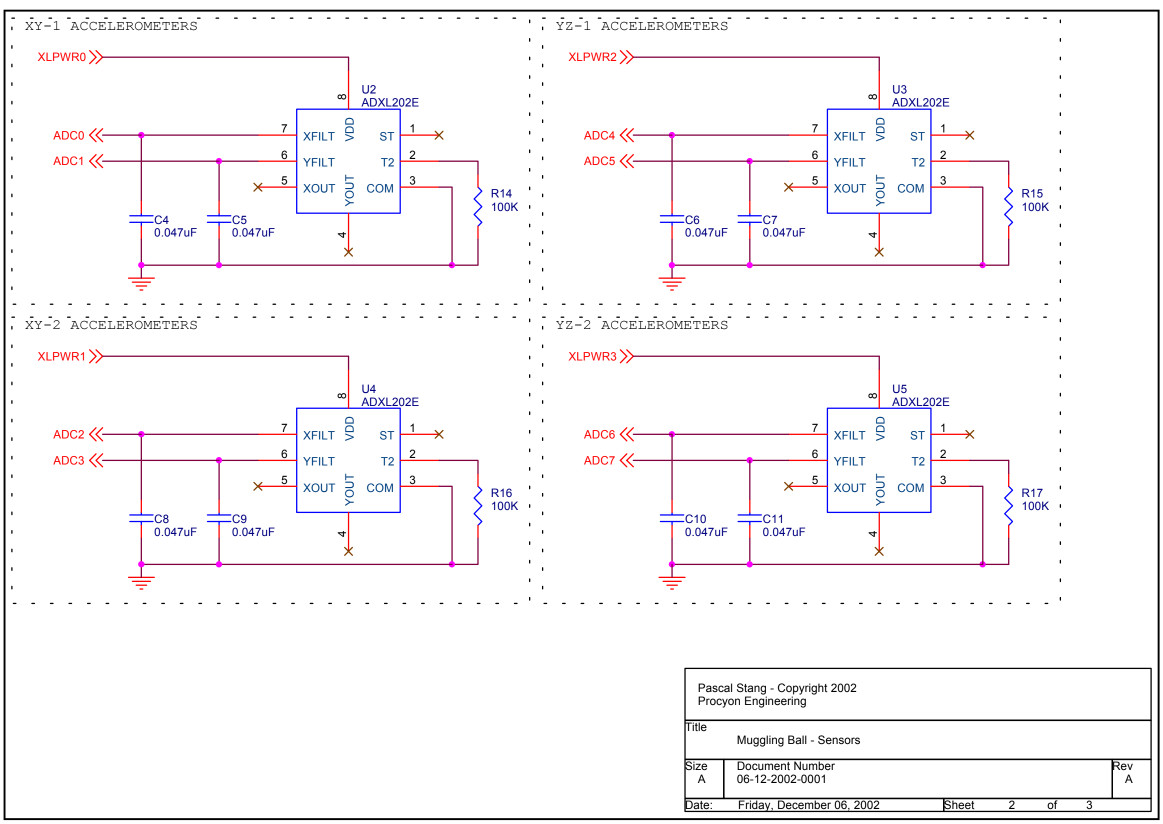
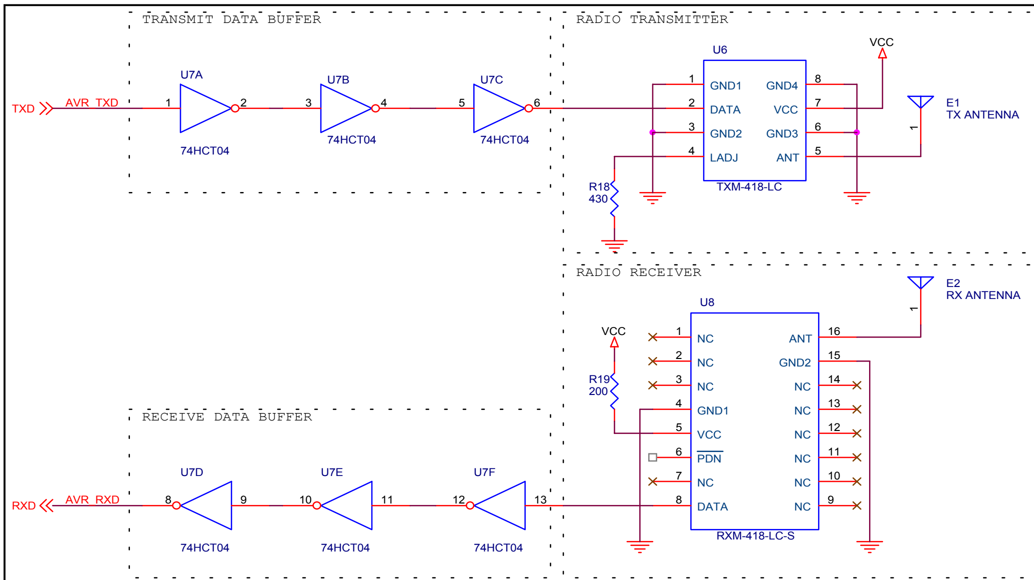


Figure 9 - chime.pd

B Schematics







Pascal Stang - Copyright 2002 Procyon Engineering		
Title: Muggling Ball - 315/418/433MHz Radio Data Transceiver		
Size: A	Document Number: 06-12-2002-0001	Rev: A
Date: Friday, December 06, 2002	Sheet 3 of 3	

C Code

C.1 Base

```
//*****************************************************************************
// File Name   : base.c
// Title              : ball receiver test code
// Revision           : 0.1
// Notes              :
// Target MCU  : Atmel AVR series
// Editor Tabs : 4
//
// Revision History:
// When               Who                   Description of change
// ----------- -----------    -----------------------
// 20-Oct-2002 pstang         Created the program
//*****************************************************************************

//----- Include Files ---------------------------------------------------------
#include <io.h>                       // include I/O definitions (port names, pin names,
etc)
#include <sig-avr.h>   // include "signal" names (interrupt names)
#include <interrupt.h> // include interrupt support
#include <progmem.h>

#include "global.h"           // include our global settings
#include "uart2.h"            // include uart function library
#include "rprintf.h"   // include printf function library
#include "timer128.h"  // include timer function library
#include "lcd.h"              // include lcd support
#include "a2d.h"              // include A/D support
#include "stxetx.h"           // include STX/ETX packet support


#define MIDI_NOTE_ON   0x90
#define MIDI_NOTE_OFF  0x80

// 1001cccc 0nnnnnnn 0vvvvvvv
#define MIDI_POLY_PRESSURE    0xA0
// 1011cccc 0nnnnnnn 0vvvvvvv
#define MIDI_CONTROL_CHANGE   0xB0
// 1100cccc 0ppppppp
#define MIDI_PROGRAM_CHANGE   0xC0

#define MIDI_DATA_MASK               0x7F
#define MIDI_STATUS_MASK             0xF0
#define MIDI_CHANNEL_MASK            0x0F

#define MIDI_BAUD_RATE               31250

struct
{
       u16 X;
       u16 Y;
       u16 Z;
       u16 Xr;
       u16 Yr;
       u16 Zr;
} Accel;

void go(void);
u08 getSw(void);
void midiNoteOnOut(u08 note, u08 vel, u08 channel);

int main(void)
{
       // initialize the AVRlib libraries
       timerInit();                          // initialize the timer system
```

```
	uartInit();				// initialize the UART (serial port)
	lcdInit();
	rprintfInit(uart1SendByte);	// init rprintf
	a2dInit();
	stxetxInit(uart0SendByte);	// init stxetx

	// set the radio comm baud rate
	uartSetBaudRate(0,4800);
	// print a debug message
	uartSetBaudRate(1,38400);
	rprintf("Base power on!\r\n");
	// set the midi comm baud rate
	uartSetBaudRate(1,38400);

	// disable RAM
	sbi(DDRC, 7);
	cbi(PORTC, 7);

	// get the lcd bias voltage set
	sbi(DDRB, 5);
	cbi(PORTB, 5);

	go();

	return 0;
}

void go(void)
{
	u08* ptr;
	u16 sample=0;

	lcdClear();

	while(1)
	{
		rprintfInit(lcdDataWrite);
		lcdGotoXY(0,0);
		rprintf("X"); lcdProgressBar(Accel.X, 256, 14); rprintfu16(Accel.X);

		lcdGotoXY(0,1);
		rprintf("Y"); lcdProgressBar(Accel.Y, 256, 14); rprintfu16(Accel.Y);

		lcdGotoXY(20,0);
		rprintf("Z"); lcdProgressBar(Accel.Z, 256, 14); rprintfu16(Accel.Z);

		lcdGotoXY(20,1);
		rprintf("Sample: "); rprintfu32(sample);

		// input STX/ETX report packet
		if(stxetxProcess(uartGetRxBuffer(0)))
		{
			// get pointer to packet data
			ptr = stxetxGetRxPacketData();
			// retrieve data values
			Accel.X = (ptr[0]<<8) + ptr[1];
			Accel.Y = (ptr[2]<<8) + ptr[3];
			Accel.Z = (ptr[4]<<8) + ptr[5];
			Accel.Xr = ptr[6];
			Accel.Yr = ptr[7];
			Accel.Zr = ptr[8];
			sample = (ptr[9]<<8) + ptr[10];

			midiNoteOnOut(Accel.X>>7,	Accel.X,	0);
			midiNoteOnOut(Accel.Y>>7,	Accel.Y,	1);
			midiNoteOnOut(Accel.Z>>7,	Accel.Z,	2);
			midiNoteOnOut(Accel.Xr>>7,	Accel.Xr,	3);
			midiNoteOnOut(Accel.Yr>>7,	Accel.Yr,	4);
			midiNoteOnOut(Accel.Zr>>7,	Accel.Zr,	5);
			midiNoteOnOut(sample>>7, sample, 6);
```

```
                        //rprintfInit(uart1SendByte);
                        //rprintf("Packet: X=%x, Y=%x, Z=%x\r\n",Accel.X,Accel.Y,Accel.Z);
                }

        }
}

void midiNoteOnOut(u08 note, u08 vel, u08 channel)
{
        uart1SendByte(MIDI_NOTE_ON | (channel & MIDI_CHANNEL_MASK));
        uart1SendByte(MIDI_DATA_MASK & note);
        uart1SendByte(MIDI_DATA_MASK & vel);
}

u08 getSw(void)
{
        u08 sw;
        // get switch status
        sw = (~inp(PINB)>>4)&0x0F;
        if(sw==4) sw=3;
        if(sw==8) sw=4;
        return sw;
}
```

C.2 Receiver Ball

```
//*****************************************************************************
// File Name   : ball.c
// Title              : ball code
// Revision           : 0.1
// Notes              :
// Target MCU  : Atmel AVR series
// Editor Tabs : 4
//
// Revision History:
// When               Who                   Description of change
// ----------- -----------    -----------------------
// 20-Oct-2002 pstang         Created the program
//*****************************************************************************

//----- Include Files ---------------------------------------------------------
#include <io.h>                     // include I/O definitions (port names, pin names,
etc)
#include <sig-avr.h>  // include "signal" names (interrupt names)
#include <interrupt.h> // include interrupt support
#include <progmem.h>

#include "global.h"           // include our global settings
#include "uart.h"             // include uart function library
#include "rprintf.h"  // include printf function library
#include "timer.h"            // include timer function library
#include "a2d.h"              // include A/D support
#include "stxetx.h"           // include STX/ETX packet support
#include "pwmcolor.h"

#define NUM_CH                8
#define FIXED_PT_BITS  4
#define INTEGER_BITS   10

#define CHX0                  1
#define CHX1                  3
#define CHY0                  4
#define CHY1                  6
#define CHZ0                  5
#define CHZ1                  7
#define CHC0                  0
#define CHC1                  2

#define ACCEL1_PWR            PB0
#define ACCEL2_PWR            PB1
#define ACCEL3_PWR            PB2
#define ACCEL4_PWR            PB3

#define LEDR_PORT             PORTB
#define LEDG_PORT             PORTC
#define LEDB_PORT             PORTC

#define LEDR_DDR              DDRB
#define LEDG_DDR              DDRC
#define LEDB_DDR              DDRC

#define LED0R                 PB4
#define LED1R                 PB5
#define LED2R                 PB6
#define LED3R                 PB7

#define LED0G                 PC4
#define LED1G                 PC5
#define LED2G                 PC6
#define LED3G                 PC7

#define LED0B                 PC0
#define LED1B                 PC1
#define LED2B                 PC2
```

```c
#define LED3B                  PC3

typedef struct
{
	s16 value;
	s32 valuefilt;
	s32 scale;
	s32 offset;
} a2dChannel;

struct
{
	a2dChannel ch[NUM_CH];
	s16 filtCoeff;
	u32 sample;
} a2dData;

struct
{
	u16 X;
	u16 Y;
	u16 Z;
	u16 Xr;
	u16 Yr;
	u16 Zr;
} Accel;

unsigned char packet[20];

void run(void);
void sample(void);

int main(void)
{
	// initialize the AVRlib libraries
	timerInit();                          // initialize the timer system
	uartInit();                           // initialize the UART (serial port)
	uartSetBaudRate(4800);
	rprintfInit(uartSendByte);    // init rprintf
	a2dInit();
	stxetxInit(uartSendByte);     // init stxetx

	// turn receiver off
	//cbi(UCSRB, RXEN);
	//cbi(UCSRB, RXCIE);

	// send a clear-text power-on message
	rprintf("\r\n\r\nBall power on!\r\n");

	// blink LEDs
	outb(LEDR_DDR, 0xFF);
	outb(LEDG_DDR, 0xFF);
	outb(LEDB_DDR, 0xFF);
	outb(LEDR_PORT, 0x00);
	outb(LEDG_PORT, 0x00);
	outb(LEDB_PORT, 0x00);
	timerPause(1000);
	outb(LEDR_PORT, 0xFF);
	outb(LEDG_PORT, 0xFF);
	outb(LEDB_PORT, 0xFF);

	run();

	return 0;
}

void run(void)
{
	u08 i=0;

	// set filters
```

```
	a2dData.filtCoeff = 10;
	a2dData.sample = 0;
	// set sensor coeffs
	a2dData.ch[0].offset = -0x05;
	a2dData.ch[0].scale = 1<<FIXED_PT_BITS;
	a2dData.ch[1].offset = 0x0B;
	a2dData.ch[1].scale = 1<<FIXED_PT_BITS;
	a2dData.ch[2].offset = 0x0A;
	a2dData.ch[2].scale = 1<<FIXED_PT_BITS;
	a2dData.ch[3].offset = -0x05;
	a2dData.ch[3].scale = 1<<FIXED_PT_BITS;
	a2dData.ch[4].offset = 0x09;
	a2dData.ch[4].scale = 1<<FIXED_PT_BITS;
	a2dData.ch[5].offset = 0x0C;
	a2dData.ch[5].scale = 1<<FIXED_PT_BITS;
	a2dData.ch[6].offset = -0x03;
	a2dData.ch[6].scale = 1<<FIXED_PT_BITS;
	a2dData.ch[7].offset = -0x0B;
	a2dData.ch[7].scale = 1<<FIXED_PT_BITS;

	// setup a2d converter
	a2dSetPrescaler(ADC_PRESCALE_DIV8);
	a2dSetReference(ADC_REFERENCE_AVCC);

	// turn on accelerometers
	sbi(DDRB,  ACCEL1_PWR);
	sbi(DDRB,  ACCEL2_PWR);
	sbi(DDRB,  ACCEL3_PWR);
	sbi(DDRB,  ACCEL4_PWR);
	sbi(PORTB, ACCEL1_PWR);
	sbi(PORTB, ACCEL2_PWR);
	sbi(PORTB, ACCEL3_PWR);
	sbi(PORTB, ACCEL4_PWR);

	// schedule sampling routine
	timer2SetPrescaler(TIMER_CLK_DIV1024);
	timerAttach(TIMER2OVERFLOW_INT, sample);

	// initialize LEDs
	timer1SetPrescaler(TIMER_CLK_DIV256);
	pwmswInit(0x0200);
	pwmswPWMSet(0, 0x0000);
	pwmswPWMSet(1, 0x0000);
	pwmswPWMSet(2, 0x0000);


	while(1)
	{
		s16 x,y,z;
		u16 c;
		// calculate linear accelerations
		Accel.X = (a2dData.ch[CHX0].valuefilt+a2dData.ch[CHX1].valuefilt)-
(0x0400<<FIXED_PT_BITS);
		Accel.Y = (a2dData.ch[CHY0].valuefilt-a2dData.ch[CHY1].valuefilt);
		Accel.Z = (a2dData.ch[CHZ0].valuefilt+a2dData.ch[CHZ1].valuefilt)-
(0x0400<<FIXED_PT_BITS);
		c = (a2dData.ch[CHC0].valuefilt-a2dData.ch[CHC1].valuefilt);

		// calculate angular accelerations
		Accel.Xr = a2dData.ch[CHX0].valuefilt-a2dData.ch[CHX1].valuefilt;
		Accel.Yr = a2dData.ch[CHY0].valuefilt+a2dData.ch[CHY1].valuefilt;
		Accel.Zr = a2dData.ch[CHZ0].valuefilt-a2dData.ch[CHZ1].valuefilt;

		// reprocess to arbitrary bit length
		x = Accel.X;
		y = Accel.Y;
		z = Accel.Z;
		x = x>>(FIXED_PT_BITS-2);
		y = y>>(FIXED_PT_BITS-2);
		z = z>>(FIXED_PT_BITS-2);
```

```
			// output STX/ETX report packet
			packet[0] = x>>8;
			packet[1] = x;
			packet[2] = y>>8;
			packet[3] = y;
			packet[4] = z>>8;
			packet[5] = z;
			packet[6] = Accel.Xr>>(FIXED_PT_BITS+3);
			packet[7] = Accel.Yr>>(FIXED_PT_BITS+3);
			packet[8] = Accel.Zr>>(FIXED_PT_BITS+3);
			packet[9] = a2dData.sample>>8;
			packet[10] = a2dData.sample;
			stxetxSend(0x00, 0x55, 11, packet);

			//timerPause(10);
			i++;

			// LED dimming
			pwmswPWMSet(0, (ABS(x)>>0) & 0x01C0);
			pwmswPWMSet(1, (ABS(y)>>0) & 0x01C0);
			pwmswPWMSet(2, (ABS(z)>>0) & 0x01C0);
			//pwmswPWMSet(0, (i<<2) & 0x1C0);
			//pwmswPWMSet(1, (i<<2) & 0x1C0);
			//pwmswPWMSet(2, (i<<2) & 0x1C0);

/*
			// LED rendering
			// red LEDs
			if(x>0)
			{
				cbi(LEDR_PORT, LED0R);
				cbi(LEDR_PORT, LED1R);
				cbi(LEDR_PORT, LED2R);
				cbi(LEDR_PORT, LED3R);
			}
			else
			{
				sbi(LEDR_PORT, LED0R);
				sbi(LEDR_PORT, LED1R);
				sbi(LEDR_PORT, LED2R);
				sbi(LEDR_PORT, LED3R);
			}

			// green LEDs
			if(y>0)
			{
				cbi(LEDG_PORT, LED0G);
				cbi(LEDG_PORT, LED1G);
				cbi(LEDG_PORT, LED2G);
				cbi(LEDG_PORT, LED3G);
			}
			else
			{
				sbi(LEDG_PORT, LED0G);
				sbi(LEDG_PORT, LED1G);
				sbi(LEDG_PORT, LED2G);
				sbi(LEDG_PORT, LED3G);
			}

			// blue LEDs
			if(z>0)
			{
				cbi(LEDB_PORT, LED0B);
				cbi(LEDB_PORT, LED1B);
				cbi(LEDB_PORT, LED2B);
				cbi(LEDB_PORT, LED3B);
			}
			else
			{
				sbi(LEDB_PORT, LED0B);
				sbi(LEDB_PORT, LED1B);
```

```
			sbi(LEDB_PORT, LED2B);
			sbi(LEDB_PORT, LED3B);
		}
*/
/*
		// channel output
		rprintf("  CH0:");	rprintfu16(a2dData.ch[0].valuefilt>>FIXED_PT_BITS);
		rprintf("  CH1:");	rprintfu16(a2dData.ch[1].valuefilt>>FIXED_PT_BITS);
		rprintf("  CH2:");	rprintfu16(a2dData.ch[2].valuefilt>>FIXED_PT_BITS);
		rprintf("  CH3:");	rprintfu16(a2dData.ch[3].valuefilt>>FIXED_PT_BITS);
		rprintf("  CH4:");	rprintfu16(a2dData.ch[4].valuefilt>>FIXED_PT_BITS);
		rprintf("  CH5:");	rprintfu16(a2dData.ch[5].valuefilt>>FIXED_PT_BITS);
		rprintf("  CH6:");	rprintfu16(a2dData.ch[6].valuefilt>>FIXED_PT_BITS);
		rprintf("  CH7:");	rprintfu16(a2dData.ch[7].valuefilt>>FIXED_PT_BITS);
		rprintfCRLF();
*/
/*		// XYZ raw output
		rprintf("  CHX0:");
	rprintfu16(a2dData.ch[CHX0].valuefilt>>FIXED_PT_BITS);
		rprintf("  CHX1:");
	rprintfu16(a2dData.ch[CHX1].valuefilt>>FIXED_PT_BITS);
		rprintf("  CHY0:");
	rprintfu16(a2dData.ch[CHY0].valuefilt>>FIXED_PT_BITS);
		rprintf("  CHY1:");
	rprintfu16(a2dData.ch[CHY1].valuefilt>>FIXED_PT_BITS);
		rprintf("  CHZ0:");
	rprintfu16(a2dData.ch[CHZ0].valuefilt>>FIXED_PT_BITS);
		rprintf("  CHZ1:");
	rprintfu16(a2dData.ch[CHZ1].valuefilt>>FIXED_PT_BITS);
		rprintf("  CHC0:");
	rprintfu16(a2dData.ch[CHC0].valuefilt>>FIXED_PT_BITS);
		rprintf("  CHC1:");
	rprintfu16(a2dData.ch[CHC1].valuefilt>>FIXED_PT_BITS);
		rprintfCRLF();
*/
/*
		// XYZ linear acceleration output
		rprintf("  X:");
		rprintfu16(Accel.X>>FIXED_PT_BITS);
		rprintf("  Y:");
		rprintfu16(Accel.Y>>FIXED_PT_BITS);
		rprintf("  Z:");
		rprintfu16(Accel.Z>>FIXED_PT_BITS);
		rprintf("  XR:");
		rprintfu16(Accel.Xr>>FIXED_PT_BITS);
		rprintf("  YR:");
		rprintfu16(Accel.Yr>>FIXED_PT_BITS);
		rprintf("  ZR:");
		rprintfu16(Accel.Zr>>FIXED_PT_BITS);
		rprintf("  P%d", i);
		rprintfCRLF();
*/
/*
		// bit-processed XYZ linear acceleration output
		rprintf("  X:");
		rprintfu16(x);
		rprintf("  Y:");
		rprintfu16(y);
		rprintf("  Z:");
		rprintfu16(z);
		rprintf("  P%d", i);
		rprintfCRLF();
*/
	}
}

void sample(void)
{
	u08 i;
	for(i=0; i<NUM_CH; i++)
	{
```

```
                // sample sensor
                a2dData.ch[i].value = a2dConvert10bit(i);
                // remove offset
                a2dData.ch[i].value += a2dData.ch[i].offset;
                // correct scale factor
                //a2dData.ch[i].value =
(a2dData.ch[i].value*a2dData.ch[i].scale)>>FIXED_PT_BITS;
                // do filter
                a2dData.ch[i].valuefilt = (a2dData.ch[i].valuefilt*(a2dData.filtCoeff-1) +
(a2dData.ch[i].value<<FIXED_PT_BITS))/a2dData.filtCoeff;
                // or don't do filter
                //a2dData.ch[i].valuefilt = a2dData.ch[i].value<<FIXED_PT_BITS;
        }
        a2dData.sample++;
}
```