// OCTOSynth-0.2

//

// Joe Marshall 2011

// Resonant filter based on Meeblip (meeblip.noisepages.com)

// Interrupt setup code based on code by Martin Nawrath (http://interface.khm.de/index.php/lab/experiments/arduino-dds-sinewave-generator/)

// oscillators and inline assembler optimisation by me.

//

// key input is from 8 capacitive inputs on digital input 6,7, and analog inputs 0-6

// each input is a single wire, going to something metal to touch

// (I used a bunch of big carriage bolts)

//

// sensing of this is done by getNoteKeys, using the method described at:

// http://www.arduino.cc/playground/Code/CapacitiveSensor

//

// I use assembler with an unrolled loop using 16 registers to detect this

// this makes things much more accurate than the C loop described on the link above

// as we are measuring the relevant delay in single processor cycles.

// It seems to be happy even with battery power, detecting up to 8 concurrent

// touches.

// The waves are all defined at the very top, because we are forcing them to align to 256 byte boundaries

// doing this makes the oscillator code quicker (as calculating a wave offset is just

// a matter of replacing the low byte of the address).

// Having said that, other arduino stuff probably gets loaded in here first

// because the aligned attribute seems to add a couple of hundred bytes to the code

#define TEST\_PATTERN\_INTRO

//#define FILTER\_LPF\_NONE

#define FILTER\_LPF\_HACK

// table of 256 sine values / one sine period / stored in flash memory

char sine256[256] \_\_attribute\_\_ ((aligned(256))) = {

0 , 3 , 6 , 9 , 12 , 15 , 18 , 21 , 24 , 27 , 30 , 33 , 36 , 39 , 42 , 45 ,

48 , 51 , 54 , 57 , 59 , 62 , 65 , 67 , 70 , 73 , 75 , 78 , 80 , 82 , 85 , 87 ,

89 , 91 , 94 , 96 , 98 , 100 , 102 , 103 , 105 , 107 , 108 , 110 , 112 , 113 , 114 , 116 ,

117 , 118 , 119 , 120 , 121 , 122 , 123 , 123 , 124 , 125 , 125 , 126 , 126 , 126 , 126 , 126 ,

127 , 126 , 126 , 126 , 126 , 126 , 125 , 125 , 124 , 123 , 123 , 122 , 121 , 120 , 119 , 118 ,

117 , 116 , 114 , 113 , 112 , 110 , 108 , 107 , 105 , 103 , 102 , 100 , 98 , 96 , 94 , 91 ,

89 , 87 , 85 , 82 , 80 , 78 , 75 , 73 , 70 , 67 , 65 , 62 , 59 , 57 , 54 , 51 ,

48 , 45 , 42 , 39 , 36 , 33 , 30 , 27 , 24 , 21 , 18 , 15 , 12 , 9 , 6 , 3 ,

0 , -3 , -6 , -9 , -12 , -15 , -18 , -21 , -24 , -27 , -30 , -33 , -36 , -39 , -42 , -45 ,

-48 , -51 , -54 , -57 , -59 , -62 , -65 , -67 , -70 , -73 , -75 , -78 , -80 , -82 , -85 , -87 ,

-89 , -91 , -94 , -96 , -98 , -100 , -102 , -103 , -105 , -107 , -108 , -110 , -112 , -113 , -114 , -116 ,

-117 , -118 , -119 , -120 , -121 , -122 , -123 , -123 , -124 , -125 , -125 , -126 , -126 , -126 , -126 , -126 ,

-127 , -126 , -126 , -126 , -126 , -126 , -125 , -125 , -124 , -123 , -123 , -122 , -121 , -120 , -119 , -118 ,

-117 , -116 , -114 , -113 , -112 , -110 , -108 , -107 , -105 , -103 , -102 , -100 , -98 , -96 , -94 , -91 ,

-89 , -87 , -85 , -82 , -80 , -78 , -75 , -73 , -70 , -67 , -65 , -62 , -59 , -57 , -54 , -51 ,

-48 , -45 , -42 , -39 , -36 , -33 , -30 , -27 , -24 , -21 , -18 , -15 , -12 , -9 , -6 , -3

};

char square256[256] \_\_attribute\_\_ ((aligned(256))) = {

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 , 127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 ,

-127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127 , -127

};

char triangle256[256] \_\_attribute\_\_ ((aligned(256))) = {

-127 , -125 , -123 , -121 , -119 , -117 , -115 , -113 , -111 , -109 , -107 , -105 , -103 , -101 , -99 , -97 ,

-95 , -93 , -91 , -89 , -87 , -85 , -83 , -81 , -79 , -77 , -75 , -73 , -71 , -69 , -67 , -65 ,

-63 , -61 , -59 , -57 , -55 , -53 , -51 , -49 , -47 , -45 , -43 , -41 , -39 , -37 , -35 , -33 ,

-31 , -29 , -27 , -25 , -23 , -21 , -19 , -17 , -15 , -13 , -11 , -9 , -7 , -5 , -3 , -1 ,

1 , 3 , 5 , 7 , 9 , 11 , 13 , 15 , 17 , 19 , 21 , 23 , 25 , 27 , 29 , 31 ,

33 , 35 , 37 , 39 , 41 , 43 , 45 , 47 , 49 , 51 , 53 , 55 , 57 , 59 , 61 , 63 ,

65 , 67 , 69 , 71 , 73 , 75 , 77 , 79 , 81 , 83 , 85 , 87 , 89 , 91 , 93 , 95 ,

97 , 99 , 101 , 103 , 105 , 107 , 109 , 111 , 113 , 115 , 117 , 119 , 121 , 123 , 125 , 127 ,

129 , 127 , 125 , 123 , 121 , 119 , 117 , 115 , 113 , 111 , 109 , 107 , 105 , 103 , 101 , 99 ,

97 , 95 , 93 , 91 , 89 , 87 , 85 , 83 , 81 , 79 , 77 , 75 , 73 , 71 , 69 , 67 ,

65 , 63 , 61 , 59 , 57 , 55 , 53 , 51 , 49 , 47 , 45 , 43 , 41 , 39 , 37 , 35 ,

33 , 31 , 29 , 27 , 25 , 23 , 21 , 19 , 17 , 15 , 13 , 11 , 9 , 7 , 5 , 3 ,

1 , -1 , -3 , -5 , -7 , -9 , -11 , -13 , -15 , -17 , -19 , -21 , -23 , -25 , -27 , -29 ,

-31 , -33 , -35 , -37 , -39 , -41 , -43 , -45 , -47 , -49 , -51 , -53 , -55 , -57 , -59 , -61 ,

-63 , -65 , -67 , -69 , -71 , -73 , -75 , -77 , -79 , -81 , -83 , -85 , -87 , -89 , -91 , -93 ,

-95 , -97 , -99 , -101 , -103 , -105 , -107 , -109 , -111 , -113 , -115 , -117 , -119 , -121 , -123 , -125

};

char sawtooth256[256] \_\_attribute\_\_ ((aligned(256))) = {

-127 , -127 , -126 , -125 , -124 , -123 , -122 , -121 , -120 , -119 , -118 , -117 , -116 , -115 , -114 , -113 ,

-112 , -111 , -110 , -109 , -108 , -107 , -106 , -105 , -104 , -103 , -102 , -101 , -100 , -99 , -98 , -97 ,

-96 , -95 , -94 , -93 , -92 , -91 , -90 , -89 , -88 , -87 , -86 , -85 , -84 , -83 , -82 , -81 ,

-80 , -79 , -78 , -77 , -76 , -75 , -74 , -73 , -72 , -71 , -70 , -69 , -68 , -67 , -66 , -65 ,

-64 , -63 , -62 , -61 , -60 , -59 , -58 , -57 , -56 , -55 , -54 , -53 , -52 , -51 , -50 , -49 ,

-48 , -47 , -46 , -45 , -44 , -43 , -42 , -41 , -40 , -39 , -38 , -37 , -36 , -35 , -34 , -33 ,

-32 , -31 , -30 , -29 , -28 , -27 , -26 , -25 , -24 , -23 , -22 , -21 , -20 , -19 , -18 , -17 ,

-16 , -15 , -14 , -13 , -12 , -11 , -10 , -9 , -8 , -7 , -6 , -5 , -4 , -3 , -2 , -1 ,

0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 ,

16 , 17 , 18 , 19 , 20 , 21 , 22 , 23 , 24 , 25 , 26 , 27 , 28 , 29 , 30 , 31 ,

32 , 33 , 34 , 35 , 36 , 37 , 38 , 39 , 40 , 41 , 42 , 43 , 44 , 45 , 46 , 47 ,

48 , 49 , 50 , 51 , 52 , 53 , 54 , 55 , 56 , 57 , 58 , 59 , 60 , 61 , 62 , 63 ,

64 , 65 , 66 , 67 , 68 , 69 , 70 , 71 , 72 , 73 , 74 , 75 , 76 , 77 , 78 , 79 ,

80 , 81 , 82 , 83 , 84 , 85 , 86 , 87 , 88 , 89 , 90 , 91 , 92 , 93 , 94 , 95 ,

96 , 97 , 98 , 99 , 100 , 101 , 102 , 103 , 104 , 105 , 106 , 107 , 108 , 109 , 110 , 111 ,

112 , 113 , 114 , 115 , 116 , 117 , 118 , 119 , 120 , 121 , 122 , 123 , 124 , 125 , 126 , 127

};

#include "avr/pgmspace.h"

// log table for 128 filter cutoffs

unsigned char logCutoffs[128] = {0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x03,0x04,0x04,0x04,0x04,0x04,0x05,0x05,0x05,0x05,0x06,0x06,0x06,0x06,0x06,0x06,0x07,0x08,0x08,0x08,0x09,0x09,0x0A,0x0A,0x0A,0x0A,0x0B,0x0C,0x0C,0x0C,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1E,0x20,0x21,0x22,0x23,0x24,0x26,0x28,0x2A,0x2C,0x2E,0x30,0x32,0x34,0x36,0x38,0x3A,0x40,0x42,0x44,0x48,0x4C,0x4F,0x52,0x55,0x58,0x5D,0x61,0x65,0x68,0x6C,0x70,0x76,0x7E,0x85,0x8A,0x90,0x96,0x9D,0xA4,0xAB,0xB0,0xBA,0xC4,0xCE,0xD8,0xE0,0xE8,0xF4,0xFF};

volatile unsigned int WAIT\_curTime;

#define WAIT\_UNTIL\_INTERRUPT() WAIT\_curTime=loopSteps; while(WAIT\_curTime==loopSteps){}

#define SERIAL\_OUT 0

// attack,decay are in 1/64ths per 125th of a second - ie. 1 = 0->1 in half a second

const int DECAY=3;

const int ATTACK=4;

volatile char\* curWave=square256;

#define cbi(sfr, bit) (\_SFR\_BYTE(sfr) &= ~\_BV(bit))

#define sbi(sfr, bit) (\_SFR\_BYTE(sfr) |= \_BV(bit))

// this is supposedly the audio clock frequency - as

// you can see, measured freq may vary a bit from supposed clock frequency

// I'm not quite sure why

// const double refclk=31372.549; // =16MHz / 510

// const double refclk=31376.6; // measured

// variables used inside interrupt service declared as voilatile

// these variables allow you to keep track of time - as delay / millis etc. are

// made inactive due to interrupts being disabled.

volatile unsigned char loopSteps=0; // once per sample

volatile unsigned int loopStepsHigh=0; // once per 256 samples

// information about the current state of a single oscillator

struct oscillatorPhase

{

unsigned int phaseStep;

char volume;

unsigned int phaseAccu;

};

// the oscillators (8 of them)

struct oscillatorPhase oscillators[8];

// tword\_m=pow(2,32)\*dfreq/refclk; // calculate DDS new tuning word

// to get hz -> tuning word do: (pow(2,16) \* frequency) / 31376.6

const unsigned int NOTE\_FREQS[53]={123,131,139,147,156,165,175,185,196,208,220,233,247,262,277,294,311,330,349,370,392,415,440,466,494,523,554,587,622,659,698,740,784,831,880,932,988,1047,1109,1175,1245,1319,1397,1480,1568,1661,1760,1865,1976,2093,2217,2349,2489};

// thresholds for the capacitive sensing buttons

int calibrationThresholds[8]={0,0,0,0,0,0,0,0};

inline int getNoteKeys(boolean calibrate=false)

{

char PORTD\_PINS=0b11000000; // (pins 6-7 - avoid pins 0,1 as they are used for serial port comms)

char PORTC\_PINS=0b111111; //(analog pins 0-5)

const int MAX\_LOOPS=16;

char port\_values[MAX\_LOOPS\*2];

WAIT\_UNTIL\_INTERRUPT();

asm volatile (

// port D reading loop:

// DDRD &= ~(PORTD\_PINS = 0x3f); // set pins 8-12 to input mode

"in %[temp],0x0a" "\n\t"

"andi %[temp],0x3f" "\n\t"

"out 0x0a,%[temp]" "\n\t"

// PORTD |= (PORTD\_PINS); // set pins 8-12 pullup on

"in %[temp],0x0b" "\n\t"

"ori %[temp],0xC0" "\n\t"

"out 0x0b,%[temp]" "\n\t"

"in %0,0x09" "\n\t"

"in %1,0x09" "\n\t"

"in %2,0x09" "\n\t"

"in %3,0x09" "\n\t"

"in %4,0x09" "\n\t"

"in %5,0x09" "\n\t"

"in %6,0x09" "\n\t"

"in %7,0x09" "\n\t"

"in %8,0x09" "\n\t"

"in %9,0x09" "\n\t"

"in %10,0x09" "\n\t"

"in %11,0x09" "\n\t"

"in %12,0x09" "\n\t"

"in %13,0x09" "\n\t"

"in %14,0x09" "\n\t"

"in %15,0x09" "\n\t"

:

// outputs

"=r" (port\_values[0]),

"=r" (port\_values[2]),

"=r" (port\_values[4]),

"=r" (port\_values[6]),

"=r" (port\_values[8]),

"=r" (port\_values[10]),

"=r" (port\_values[12]),

"=r" (port\_values[14]),

"=r" (port\_values[16]),

"=r" (port\_values[18]),

"=r" (port\_values[20]),

"=r" (port\_values[22]),

"=r" (port\_values[24]),

"=r" (port\_values[26]),

"=r" (port\_values[28]),

"=r" (port\_values[30])

:[temp] "d" (0));

WAIT\_UNTIL\_INTERRUPT();

asm volatile (

// port C reading loop:

// DDRC &= ~(PORTC\_PINS = 0xc0); // set pins 5-7 to input mode

"in %[temp],0x07" "\n\t"

"andi %[temp],0xc0" "\n\t"

"out 0x07,%[temp]" "\n\t"

// PORTC |= (PORTC\_PINS); // set pins 5-7 pullup on

"in %[temp],0x08" "\n\t"

"ori %[temp],0x3F" "\n\t"

"out 0x08,%[temp]" "\n\t"

"in %0,0x06" "\n\t"

"in %1,0x06" "\n\t"

"in %2,0x06" "\n\t"

"in %3,0x06" "\n\t"

"in %4,0x06" "\n\t"

"in %5,0x06" "\n\t"

"in %6,0x06" "\n\t"

"in %7,0x06" "\n\t"

"in %8,0x06" "\n\t"

"in %9,0x06" "\n\t"

"in %10,0x06" "\n\t"

"in %11,0x06" "\n\t"

"in %12,0x06" "\n\t"

"in %13,0x06" "\n\t"

"in %14,0x06" "\n\t"

"in %15,0x06" "\n\t"

:

// outputs

"=r" (port\_values[1]),

"=r" (port\_values[3]),

"=r" (port\_values[5]),

"=r" (port\_values[7]),

"=r" (port\_values[9]),

"=r" (port\_values[11]),

"=r" (port\_values[13]),

"=r" (port\_values[15]),

"=r" (port\_values[17]),

"=r" (port\_values[19]),

"=r" (port\_values[21]),

"=r" (port\_values[23]),

"=r" (port\_values[25]),

"=r" (port\_values[27]),

"=r" (port\_values[29]),

"=r" (port\_values[31])

:[temp] "d" (0));

PORTC &= ~(PORTC\_PINS); // pullup off pins 8-12

PORTD &= ~(PORTD\_PINS); // pullup off pins 5-7

DDRC |= (PORTC\_PINS); // discharge

DDRD |= (PORTD\_PINS); // discharge

if(calibrate)

{

for(int c=0;c<8;c++)

{

for(int d=0;d<MAX\_LOOPS;d++)

{

int liveNotes=((int\*)port\_values)[d];

liveNotes&=0x3fc0;

liveNotes>>=6;

if(liveNotes&(1<<c))

{

if(calibrationThresholds[c]<=d)

{

calibrationThresholds[c]=d+1;

}

break;

}

}

}

}

int liveNotes=0;

for(int c=0;c<8;c++)

{

int val = ((int\*)port\_values)[calibrationThresholds[c]+1];

val&=0x3fc0;

val>>=6;

if((val&(1<<c))==0)

{

liveNotes|=(1<<c);

}

}

return liveNotes;

}

// get capacitive touch on input 4 and output 3

// used for filter modulator

inline int getfiltermodulationtime()

{

static int running\_average=0;

static int running\_min=1024;

static int running\_min\_inc\_count=0;

static boolean initialise\_running\_min=true;

unsigned int delayTime=0;

char PINNUM\_OUT=3;

char PINNUM\_IN=4;

char PIN\_OUT=1<<PINNUM\_OUT;

char PIN\_IN=1<<PINNUM\_IN;

// make sure inputs / outputs are set right

DDRD|=PIN\_OUT;

DDRD&=~(PIN\_IN);

WAIT\_UNTIL\_INTERRUPT();

PORTD|=PIN\_OUT;

asm volatile (

"loopstart%=:" "\n\t"

"sbic 0x09,%[PINNUM\_IN]" "\n\t"

"rjmp outloop%=" "\n\t"

"adiw %[delayTime],0x01" "\n\t"

"cpi %B[delayTime],0x02" "\n\t"

"brne loopstart%=" "\n\t"

"outloop%=:" "\n\t"

:[delayTime] "+&w" (delayTime)

:[PINNUM\_IN] "I" (PINNUM\_IN));

// set pin down - maybe don't bother timing, if it doesn't seem to add

// much accuracy?

WAIT\_UNTIL\_INTERRUPT();

PORTD&=~PIN\_OUT;

asm(

"loopstart%=:" "\n\t"

"sbis 0x09,%[PINNUM\_IN]" "\n\t"

"rjmp outloop%=" "\n\t"

"adiw %[delayTime],0x01" "\n\t"

"cpi %B[delayTime],0x02" "\n\t"

"brne loopstart%=" "\n\t"

"outloop%=:" "\n\t"

:[delayTime] "+&w" (delayTime)

:[PINNUM\_IN] "I" (PINNUM\_IN));

running\_average=(running\_average-(running\_average>>4))+(delayTime>>4);

running\_min\_inc\_count++;

if(running\_min\_inc\_count==255)

{

if(initialise\_running\_min)

{

running\_min=running\_average;

running\_min\_inc\_count=0;

initialise\_running\_min=false;

}else{

running\_min\_inc\_count=0;

running\_min++;

}

}

if(running\_average<running\_min)

{

running\_min=running\_average;

}

int touchVal=running\_average-running\_min;

if(touchVal>15)

{

touchVal-=15;

if(touchVal>99)

{

touchVal=99;

}

}else{

touchVal=0;

}

return touchVal;

}

// get capacitive touch on input 5 and output 3

// used for pitch bend

inline int getpitchbendtime()

{

static int running\_average=0;

static int running\_min=1024;

static int running\_min\_inc\_count=0;

static boolean initialise\_running\_min=true;

unsigned int delayTime=0;

char PINNUM\_OUT=3;

char PINNUM\_IN=5;

char PIN\_OUT=1<<PINNUM\_OUT;

char PIN\_IN=1<<PINNUM\_IN;

// make sure inputs / outputs are set right

DDRD|=PIN\_OUT;

DDRD&=~(PIN\_IN);

WAIT\_UNTIL\_INTERRUPT();

PORTD|=PIN\_OUT;

asm volatile (

"loopstart%=:" "\n\t"

"sbic 0x09,%[PINNUM\_IN]" "\n\t"

"rjmp outloop%=" "\n\t"

"adiw %[delayTime],0x01" "\n\t"

"cpi %B[delayTime],0x02" "\n\t"

"brne loopstart%=" "\n\t"

"outloop%=:" "\n\t"

:[delayTime] "+&w" (delayTime)

:[PINNUM\_IN] "I" (PINNUM\_IN));

// set pin down - maybe don't bother timing, if it doesn't seem to add

// much accuracy?

WAIT\_UNTIL\_INTERRUPT();

PORTD&=~PIN\_OUT;

asm(

"loopstart%=:" "\n\t"

"sbis 0x09,%[PINNUM\_IN]" "\n\t"

"rjmp outloop%=" "\n\t"

"adiw %[delayTime],0x01" "\n\t"

"cpi %B[delayTime],0x02" "\n\t"

"brne loopstart%=" "\n\t"

"outloop%=:" "\n\t"

:[delayTime] "+&w" (delayTime)

:[PINNUM\_IN] "I" (PINNUM\_IN));

running\_average=(running\_average-(running\_average>>4))+(delayTime>>4);

running\_min\_inc\_count++;

if(running\_min\_inc\_count==255)

{

if(initialise\_running\_min)

{

running\_min=running\_average;

running\_min\_inc\_count=0;

initialise\_running\_min=false;

}else{

running\_min\_inc\_count=0;

running\_min++;

}

}

if(running\_average<running\_min)

{

running\_min=running\_average;

}

int touchVal=running\_average-running\_min;

if(touchVal>15)

{

touchVal-=15;

if(touchVal>99)

{

touchVal=99;

}

}else{

touchVal=0;

}

return touchVal;

}

unsigned int pitchBendTable[201]={241, 241, 241, 242, 242, 242, 242, 242, 242, 242, 243, 243, 243, 243, 243, 243, 243, 244, 244, 244, 244, 244, 244, 244, 245, 245, 245, 245, 245, 245, 245, 246, 246, 246, 246, 246, 246, 246, 247, 247, 247, 247, 247, 247, 247, 248, 248, 248, 248, 248, 248, 248, 249, 249, 249, 249, 249, 249, 249, 250, 250, 250, 250, 250, 250, 250, 251, 251, 251, 251, 251, 251, 251, 252, 252, 252, 252, 252, 252, 253, 253, 253, 253, 253, 253, 253, 254, 254, 254, 254, 254, 254, 254, 255, 255, 255, 255, 255, 255, 256,

256,256, 256, 256, 256, 256, 256, 256, 257, 257, 257, 257, 257, 257, 257, 258, 258, 258, 258, 258, 258, 259, 259, 259, 259, 259, 259, 259, 260, 260, 260, 260, 260, 260, 260, 261, 261, 261, 261, 261, 261, 262, 262, 262, 262, 262, 262, 262, 263, 263, 263, 263, 263, 263, 264, 264, 264, 264, 264, 264, 264, 265, 265, 265, 265, 265, 265, 266, 266, 266, 266, 266, 266, 266, 267, 267, 267, 267, 267, 267, 268, 268, 268, 268, 268, 268, 269, 269, 269, 269, 269, 269, 269, 270, 270, 270, 270, 270, 270, 271, 271};

void setupNoteFrequencies(int baseNote,int pitchBendVal /\*-100 -> 100\*/)

{

oscillators[0].phaseStep=NOTE\_FREQS[baseNote-10];

oscillators[1].phaseStep=NOTE\_FREQS[baseNote-9];

oscillators[2].phaseStep=NOTE\_FREQS[baseNote-8];

oscillators[3].phaseStep=NOTE\_FREQS[baseNote-7];

oscillators[4].phaseStep=NOTE\_FREQS[baseNote-6];

oscillators[5].phaseStep=NOTE\_FREQS[baseNote-5];

oscillators[6].phaseStep=NOTE\_FREQS[baseNote-4];

oscillators[7].phaseStep=NOTE\_FREQS[baseNote-3];

// baseNote je 13-sty ton

if(pitchBendVal<-99)

{

pitchBendVal=-99;

}else if(pitchBendVal>99)

{

pitchBendVal=99;

}

// Serial.print("\*");

// Serial.print(pitchBendVal);

unsigned int pitchBendMultiplier=pitchBendTable[pitchBendVal+100];

// Serial.print(":");

// Serial.print(pitchBendMultiplier);

for(int c=0;c<8;c++)

{

// multiply 2 16 bit numbers together and shift 8 without precision loss

// requires assembler really

volatile unsigned char zeroReg=0;

volatile unsigned int multipliedCounter=oscillators[c].phaseStep;

asm volatile

(

// high bytes mult together = high byte

"ldi %A[outVal],0" "\n\t"

"mul %B[phaseStep],%B[pitchBend]" "\n\t"

"mov %B[outVal],r0" "\n\t"

// ignore overflow into r1 (should never overflow)

// low byte \* high byte -> both bytes

"mul %A[phaseStep],%B[pitchBend]" "\n\t"

"add %A[outVal],r0" "\n\t"

// carry into high byte

"adc %B[outVal],r1" "\n\t"

// high byte\* low byte -> both bytes

"mul %B[phaseStep],%A[pitchBend]" "\n\t"

"add %A[outVal],r0" "\n\t"

// carry into high byte

"adc %B[outVal],r1" "\n\t"

// low byte \* low byte -> round

"mul %A[phaseStep],%A[pitchBend]" "\n\t"

// the adc below is to round up based on high bit of low\*low:

"adc %A[outVal],r1" "\n\t"

"adc %B[outVal],%[ZERO]" "\n\t"

"clr r1" "\n\t"

:[outVal] "=&d" (multipliedCounter)

:[phaseStep] "d" (oscillators[c].phaseStep),[pitchBend] "d"( pitchBendMultiplier),[ZERO] "d" (zeroReg)

:"r1","r0"

);

oscillators[c].phaseStep=multipliedCounter;

}

// Serial.print(":");

// Serial.print(NOTE\_FREQS[baseNote]);

// Serial.print(":");

// Serial.println(oscillators[0].phaseStep);

}

void setup()

{

Serial.begin(9600); // connect to the serial port

#ifndef FILTER\_LPF\_NONE

setFilter(127, 0);

#endif

pinMode(11, OUTPUT); // pin11= PWM output / frequency output

setupNoteFrequencies(12,0);

for(int c=0;c<8;c++)

{

oscillators[c].volume=0;

}

Setup\_timer2();

// disable interrupts to avoid timing distortion

cbi (TIMSK0,TOIE0); // disable Timer0 !!! delay() is now not available

sbi (TIMSK2,TOIE2); // enable Timer2 Interrupt

// calibrate the unpressed key values

for(int x=0;x<1024;x++)

{

getNoteKeys(true);

int steps=loopSteps;

WAIT\_UNTIL\_INTERRUPT();

// int afterSteps=loopSteps;

// Serial.println(afterSteps-steps);

}

// test pattern intro

#ifdef TEST\_PATTERN\_INTRO

int filtValue=255;

byte notes[]={0x1};

for(int note=0;note<sizeof(notes)/sizeof(byte);note++)

{

int noteCount=0;

for(int c=0;c<8;c++)

{

if(notes[note]&(1<<c))

{

noteCount+=1;

}

}

for(int c=0;c<8;c++)

{

if(notes[note]&(1<<c))

{

// tu sa da stisit uvodna znelka

oscillators[c].volume=0/noteCount;

}else

{

oscillators[c].volume=0;

}

}

for(int c=0;c<50;c++)

{

// might as well keep calibrating here

// nb: each calibration loop = at least 1 interrupt

getNoteKeys(true);

#ifndef FILTER\_LPF\_NONE

setFilter(127-c, 64);

#endif

}

}

#else

// just beep to show calibration is done

oscillators[0].volume=63;

for(int c=0;c<20;c++)

{

WAIT\_UNTIL\_INTERRUPT();

}

oscillators[0].volume=63;

#endif

Serial.println("Calibrations:");

for(int c=0;c<8;c++)

{

Serial.print(c);

Serial.print(":");

Serial.println(calibrationThresholds[c]);

}

}

void loop()

{

// we keep a list of 'raw' volumes - and turn down the volume if a chord is taking >64 volume total

// this is to allow chording without reducing the volume of single notes

int rawVolumes[8]={0,0,0,0,0,0,0,0};

int curNote=0;

unsigned int filterSweep=64;

const int MIN\_SWEEP=64;

const int MAX\_SWEEP=127;

const int SWEEP\_SPEED=3;

int sweepDir=SWEEP\_SPEED;

unsigned int lastStep=loopStepsHigh;

unsigned curStep=loopStepsHigh;

while(1)

{

lastStep=curStep;

curStep=loopStepsHigh;

// NOTE: timers do not work in this code (interrupts disabled / speeds changed), so don't even think about calling: delay(), millis / micros etc.

// each loopstep is roughly 31250 / second

// this main loop will get called once every 3 or 4 samples if the serial output is turned off, maybe slower otherwise

int liveNotes=getNoteKeys();

// we are right after an interrupt (as loopStep has just been incremented)

// so we should have enough time to do the capacitative key checks

if(lastStep!=curStep)

{

int totalVolume=0;

for(int c=0;c<8;c++)

{

if((liveNotes&(1<<c))==0)

{

rawVolumes[c]-=DECAY\*(curStep-lastStep);

if(rawVolumes[c]<0)rawVolumes[c]=0;

if(SERIAL\_OUT)Serial.print(".");

}

else

{

rawVolumes[c]+=ATTACK\*(curStep-lastStep);

if(rawVolumes[c]>63)rawVolumes[c]=63;

if(SERIAL\_OUT)Serial.print(c);

}

totalVolume+=rawVolumes[c];

}

WAIT\_UNTIL\_INTERRUPT();

if( totalVolume<64 )

{

for(int c=0;c<8;c++)

{

oscillators[c].volume=rawVolumes[c];

}

}else

{

// total volume too much, scale down to avoid clipping

for(int c=0;c<8;c++)

{

oscillators[c].volume=(rawVolumes[c]\*63)/totalVolume;

}

}

}

if(SERIAL\_OUT)Serial.println("");

#ifndef FILTER\_LPF\_NONE

/\* if(liveNotes==0)

{

filterSweep=64;

sweepDir=SWEEP\_SPEED;

}

filterSweep+=sweepDir;

if(filterSweep>=MAX\_SWEEP)

{

filterSweep=MAX\_SWEEP;

sweepDir=-sweepDir;

}

else if (filterSweep<=MIN\_SWEEP)

{

sweepDir=-sweepDir;

filterSweep=MIN\_SWEEP;

}\*/

// Serial.println((int)filterValue);

// filterSweep=127-(getpitchbendtime()>>1);

WAIT\_UNTIL\_INTERRUPT();

setFilter(150-(getfiltermodulationtime()),220);

#endif

WAIT\_UNTIL\_INTERRUPT();

setupNoteFrequencies(12,-getpitchbendtime());

// we are right after an interrupt again (as loopStep has just been incremented)

// so we should have enough time to check the pitch bend capacitance without going over another sample, timing is quite important here

// need to balance using a big enough resistor to get decent sensing distance with taking too long to sample

// check the pitch bend input

}

}

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

// timer2 setup

// set prscaler to 1, PWM mode to phase correct PWM, 16000000/510 = 31372.55 Hz clock

void Setup\_timer2() {

// Timer2 Clock Prescaler to : 1

sbi (TCCR2B, CS20);

cbi (TCCR2B, CS21);

cbi (TCCR2B, CS22);

// Timer2 PWM Mode set to Phase Correct PWM

cbi (TCCR2A, COM2A0); // clear Compare Match

sbi (TCCR2A, COM2A1);

sbi (TCCR2A, WGM20); // Mode 1 / Phase Correct PWM

cbi (TCCR2A, WGM21);

cbi (TCCR2B, WGM22);

}

#ifdef FILTER\_LPF\_BIQUAD

char filtValueA1=0,filtValueA2=0,filtValueA3=0,filtValueB1=0,filtValueB2=0;

volatile unsigned char filtCoeffA1=255;

volatile char filtCoeffB1=127;

volatile unsigned char filtCoeffB2=255;

#endif

#ifdef FILTER\_LPF\_HACK

// hacked low pass filter - 2 pole resonant -

// a += f\*((in-a)+ q\*(a-b)

// b+= f\* (a-b)

int filterA=0;

int filterB=0;

unsigned char filterQ=0;

unsigned char filterF=255;

inline void setFilterRaw(unsigned char filterF, unsigned char resonance)

{

unsigned char tempReg=0,tempReg2=0;

asm volatile("ldi %[tempReg], 0xff" "\n\t"

"sub %[tempReg],%[filtF] " "\n\t"

"lsr %[tempReg]" "\n\t"

"ldi %[tempReg2],0x04" "\n\t"

"add %[tempReg],%[tempReg2]" "\n\t"

"sub %[reso],%[tempReg]" "\n\t"

"brcc Res\_Overflow%=" "\n\t"

"ldi %[reso],0x00" "\n\t"

"Res\_Overflow%=:" "\n\t"

"mov %[filtQ],%[reso]" "\n\t"

:[tempReg] "=&d" (tempReg),[tempReg2] "=&d" (tempReg2),[filtQ] "=&d" (filterQ): [reso] "d" (resonance), [filtF] "d" (filterF) );

}

inline void setFilter(unsigned char f, unsigned char resonance)

{

if(f>127)f=127;

filterF=logCutoffs[f];

setFilterRaw(filterF,resonance);

}

#endif

#define HIBYTE(\_\_x) ((char)(((unsigned int)\_\_x)>>8))

#define LOBYTE(\_\_x) ((char)(((unsigned int)\_\_x)&0xff))

// oscillator main loop (increment wavetable pointer, and add it to the output registers)

// 13 instructions - should take 14 processor cycles according to the datasheet

// in theory I think this means that each oscillator should take 1.5% of cpu

// (plus a constant overhead for interrupt calls etc.)

// Note: this used to do all the stepvolume loads near the start, but they are now interleaved in the

// code, this is because ldd (load with offset) takes 2 instructions,

// versus ld,+1 (load with post increment) and st,+1 which are 1 instruction - we can do this because:

//

// a)the step (which doesn't need to be stored back) is in memory before the

// phase accumulator (which does need to be stored back once the step is added

//

// b)the phase assumulator is stored in low byte, high byte order, meaning that we

// can add the first bytes together, then store that byte incrementing the pointer,

// then load the high byte, add the high bytes together and store incrementing the pointer

//

// I think this is the minimum number of operations possible to code this oscillator in, because

// 1)There are 6 load operations required (to load stepHigh/Low,phaseH/L,volume, and the value from the wave)

// 2)There are 2 add operations required to add to the phase accumulator

// 3)There are 2 store operations required to save the phase accumulator

// 4)There is 1 multiply (2 instruction cycles) required to do the volume

// 5)There are 2 add operations required to add to the final output

//

// 6+2+2+2+2 = 14 instruction cycles

#define OSCILLATOR\_ASM \

/\* load phase step and volume\*/ \

"ld %A[tempStep],%a[stepVolume]+" "\n\t" \

"ld %B[tempStep],%a[stepVolume]+" "\n\t" \

"ld %[tempVolume],%a[stepVolume]+" "\n\t" \

/\* load phase accumulator - high byte goes straight\*/ \

/\* into the wave lookup array (wave is on 256 byte boundary\*/ \

/\* so we can do this without any adds \*/ \

/\* Do the phase adds in between the two loads, \*/\

/\*as load with offset is slower than just a normal load \*/\

"ld %A[tempPhaseLow],%a[stepVolume]" "\n\t" \

/\* add phase step low \*/ \

"add %[tempPhaseLow],%A[tempStep]" "\n\t"\

/\* store phase accumulator low \*/ \

"st %a[stepVolume]+,%[tempPhaseLow]" "\n\t" \

/\* load phase accumulator high\*/\

"ld %A[waveBase],%a[stepVolume]" "\n\t" \

/\* add phase step high - with carry from the add above \*/\

"adc %A[waveBase],%B[tempStep]" "\n\t"\

/\* store phase step high \*/\

"st %a[stepVolume]+,%A[waveBase]" "\n\t" \

/\* now lookup from the wave - high byte = wave pointer, low byte=offset\*/ \

"ld %[tempPhaseLow],%a[waveBase]" "\n\t" \

/\* now multiply by volume\*/ \

"muls %[tempPhaseLow],%[tempVolume]" "\n\t" \

/\* r0 now contains a sample - add it to output value\*/ \

"add %A[outValue],r0" "\n\t" \

"adc %B[outValue],r1" "\n\t" \

/\* go to next oscillator - stepVolume is pointing at next\*/ \

/\* oscillator already \*/ \

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

// Timer2 Interrupt Service at 31372,550 KHz = 32uSec

// this is the timebase REFCLOCK for the DDS generator

// FOUT = (M (REFCLK)) / (2 exp 32)

// runtime : ?

ISR(TIMER2\_OVF\_vect) {

// now set up the next value

// this loop takes roughly 172 cycles (214 including the push/pops) - we have 510, so roughly 50% of the processor going spare for non-audio tasks

// the low pass filter also takes some cycles

int outValue;

// pointers:

// X = oscillator phase accumulator

// Y = oscillator step and volume

// Z = wave pos - needs to add to base

int tempStep=0;

char tempPhaseLow=0,tempVolume=0;

int tempWaveBase=0;

asm volatile (

"ldi %A[outValue],0" "\n\t"

"ldi %B[outValue],0" "\n\t"

// oscillator 0

// uncomment the code below to check

// that registers aren't getting double assigned

/\* "lds %A[outValue],0x00" "\n\t"

"lds %B[outValue],0x01" "\n\t"

"lds %A[tempPhaseLow],0x02" "\n\t"

// "lds %B[tempPhase],0x03" "\n\t"

"lds %A[tempStep],0x04" "\n\t"

"lds %B[tempStep],0x05" "\n\t"

"lds %[tempVolume],0x06" "\n\t"

"lds %[ZERO],0x07" "\n\t"

"lds %A[tempWaveBase],0x08" "\n\t"

"lds %B[tempWaveBase],0x09" "\n\t"

"lds %A[phaseAccu],0x0a" "\n\t"

"lds %B[phaseAccu],0x0b" "\n\t"

"lds %A[stepVolume],0x0c" "\n\t"

"lds %B[stepVolume],0x0d" "\n\t"

"lds %A[waveBase],0x0e" "\n\t"

"lds %B[waveBase],0x0f" "\n\t"\*/

OSCILLATOR\_ASM

OSCILLATOR\_ASM

OSCILLATOR\_ASM

OSCILLATOR\_ASM

OSCILLATOR\_ASM

OSCILLATOR\_ASM

OSCILLATOR\_ASM

OSCILLATOR\_ASM

:

// outputs

[tempPhaseLow] "=&d" (tempPhaseLow),

[tempStep] "=&d" (tempStep),

[tempVolume] "=&d" (tempVolume),

[outValue] "=&d" (outValue)

:

// inputs

[stepVolume] "y" (&oscillators[0].phaseStep),

[waveBase] "z" (256\*(((unsigned int)curWave)>>8))

:

// other registers we clobber (by doing multiplications)

"r1"

);

// at this point outValue = oscillator value

// it is currently maxed to full volume / 4

// to allow some headroom for filtering

#ifdef FILTER\_LPF\_HACK

// a low pass filter based on the one from MeeBlip (http://meeblip.noisepages.com)

// a += f\*((in-a)+ q\*(a-b)

// b+= f\* (a-b)

// outValue>>=3;

// started at 4700

// 4686

int tempReg,tempReg2=0;

unsigned char zeroRegFilt=0;

// de-volatilisati

unsigned char filtF=filterF;

unsigned char filtQ=filterQ;

asm volatile(

"sub %A[outVal],%A[filtA]" "\n\t"

"sbc %B[outVal],%B[filtA]" "\n\t"

"brvc No\_overflow1%=" "\n\t"

"ldi %A[outVal],0b00000001" "\n\t"

"ldi %B[outVal],0b10000000" "\n\t"

"No\_overflow1%=:" "\n\t"

// outVal = (in - filtA)

"mov %A[tempReg],%A[filtA]" "\n\t"

"mov %B[tempReg],%B[filtA]" "\n\t"

"sub %A[tempReg],%A[filtB]" "\n\t"

"sbc %B[tempReg],%B[filtB]" "\n\t"

"brvc No\_overflow3%=" "\n\t"

"ldi %A[tempReg],0b00000001" "\n\t"

"ldi %B[tempReg],0b10000000" "\n\t"

"No\_overflow3%=:" "\n\t"

// tempReg = (a-b)

"mulsu %B[tempReg],%[filtQ]" "\n\t"

"movw %A[tempReg2],r0" "\n\t"

// tempReg2 = (HIBYTE(a-b))\*Q

"mul %A[tempReg],%[filtQ]" "\n\t"

"add %A[tempReg2],r1" "\n\t"

"adc %B[tempReg2],%[ZERO]" "\n\t"

"rol r0" "\n\t"

"brcc No\_Round1%=" "\n\t"

"inc %A[tempReg2]" "\n\t"

"No\_Round1%=:" "\n\t"

// at this point tempReg2 = (a-b)\*Q (shifted appropriately and rounded)

// "clc" "\n\t"

"lsl %A[tempReg2]" "\n\t"

"rol %B[tempReg2]" "\n\t"

// "clc" "\n\t"

"lsl %A[tempReg2]" "\n\t"

"rol %B[tempReg2]" "\n\t"

// tempReg2 = (a-b)\*Q\*4

"add %A[outVal],%A[tempReg2]" "\n\t"

"adc %B[outVal],%B[tempReg2]" "\n\t"

"brvc No\_overflow4%=" "\n\t"

"ldi %A[outVal],0b11111111" "\n\t"

"ldi %B[outVal],0b01111111" "\n\t"

"No\_overflow4%=:" "\n\t"

// outVal = ((in-a) + (a-b)\*(Q>>8)\*4) - clipped etc

"mulsu %B[outVal],%[filtF]" "\n\t"

"movw %A[tempReg],r0" "\n\t"

"mul %A[outVal],%[filtF]" "\n\t"

"add %A[tempReg],r1" "\n\t"

"adc %B[tempReg],%[ZERO]" "\n\t"

"rol r0" "\n\t"

"brcc No\_Round2%=" "\n\t"

"inc %A[tempReg]" "\n\t"

// tempReg = f\* ((in-a) + (a-b)\*(Q>>8)\*4)

"No\_Round2%=:" "\n\t"

"add %A[filtA],%A[tempReg]" "\n\t"

"adc %B[filtA],%B[tempReg]" "\n\t"

// A= A+ f\* ((in-a) + (a-b)\*(Q>>8)\*4)

"brvc No\_overflow5%=" "\n\t"

"ldi %A[outVal],0b11111111" "\n\t"

"ldi %B[outVal],0b01111111" "\n\t"

"No\_overflow5%=:" "\n\t"

// now calculate B= f\* (a - b)

"mov %A[tempReg],%A[filtA]" "\n\t"

"mov %B[tempReg],%B[filtA]" "\n\t"

"sub %A[tempReg],%A[filtB]" "\n\t"

"sbc %B[tempReg],%B[filtB]" "\n\t"

"brvc No\_overflow6%=" "\n\t"

"ldi %A[tempReg],0b00000001" "\n\t"

"ldi %B[tempReg],0b10000000" "\n\t"

"No\_overflow6%=:" "\n\t"

// tempReg = (a-b)

"mulsu %B[tempReg],%[filtF]" "\n\t"

"movw %A[tempReg2],r0" "\n\t"

"mul %A[tempReg],%[filtF]" "\n\t"

"add %A[tempReg2],r1" "\n\t"

"adc %B[tempReg2],%[ZERO]" "\n\t"

// tempReg2 = f\*(a-b)

"add %A[filtB],%A[tempReg2]" "\n\t"

"adc %B[filtB],%B[tempReg2]" "\n\t"

"brvc No\_overflow7%=" "\n\t"

"ldi %A[filtB],0b11111111" "\n\t"

"ldi %B[filtB],0b01111111" "\n\t"

"No\_overflow7%=:" "\n\t"

// now b= b+f\*(a-b)

"mov %A[outVal],%A[filtB]" "\n\t"

"mov %B[outVal],%B[filtB]" "\n\t"

// multiply outval by 4 and clip

"add %A[outVal],%A[filtB]" "\n\t"

"adc %B[outVal],%B[filtB]" "\n\t"

"brbs 3, Overflow\_End%=" "\n\t"

"add %A[outVal],%A[filtB]" "\n\t"

"adc %B[outVal],%B[filtB]" "\n\t"

"brbs 3, Overflow\_End%=" "\n\t"

"add %A[outVal],%A[filtB]" "\n\t"

"adc %B[outVal],%B[filtB]" "\n\t"

"brbs 3, Overflow\_End%=" "\n\t"

"rjmp No\_overflow%=" "\n\t"

"Overflow\_End%=:" "\n\t"

"brbs 2,Overflow\_High%=" "\n\t"

"ldi %A[outVal],0b00000001" "\n\t"

"ldi %B[outVal],0b10000000" "\n\t"

"rjmp No\_overflow%=" "\n\t"

"Overflow\_High%=:" "\n\t"

"ldi %A[outVal],0b11111111" "\n\t"

"ldi %B[outVal],0b01111111" "\n\t"

"No\_overflow%=:" "\n\t"

//char valOut=((unsigned int)(outValue))>>8;

//valOut+=128;

// OCR2A=(byte)valOut;

"subi %B[outVal],0x80" "\n\t"

"sts 0x00b3,%B[outVal]" "\n\t"

// uncomment the lines below to see the register allocations

/\*

"lds %A[filtA],0x01" "\n\t"

"lds %B[filtA],0x02" "\n\t"

"lds %A[filtB],0x03" "\n\t"

"lds %B[filtB],0x04" "\n\t"

"lds %[filtQ],0x05" "\n\t"

"lds %[filtF],0x06" "\n\t"

"lds %A[outVal],0x07" "\n\t"

"lds %B[outVal],0x08" "\n\t"

"lds %A[tempReg],0x09" "\n\t"

"lds %B[tempReg],0x0a" "\n\t"

"lds %A[tempReg2],0x0b" "\n\t"

"lds %B[tempReg2],0x0c" "\n\t"

"lds %[ZERO],0x0d" "\n\t"\*/

:

// outputs / read/write arguments

[filtA] "+&w" (filterA),

[filtB] "+&w" (filterB),

[tempReg] "=&a" (tempReg),

[tempReg2] "=&d" (tempReg2)

:

[filtQ] "a" (filtQ),

[filtF] "a" (filtF),

[outVal] "a" (outValue),

[ZERO] "d" (zeroRegFilt)

// inputs

: "r1");

#endif

// output is done in the filter assembler code if filters are on

// otherwise we output it by hand here

#ifdef FILTER\_LPF\_NONE

// full gain

outValue\*=4;

// at this point, outValue is a 16 bit signed version of what we want ie. 0 -> 32767, then -32768 -> -1 (0xffff)

// we want 0->32767 to go to 32768-65535 and -32768 -> -1 to go to 0-32767, then we want only the top byte

// take top byte, then add 128, then cast to unsigned. The (unsigned int) in the below is to avoid having to shift (ie.just takes top byte)

char valOut=((unsigned int)(outValue))>>8;

valOut+=128;

OCR2A=(byte)valOut;

#endif

// increment loop step counter (and high counter)

// these are used because we stop the timer

// interrupt running, so have no other way to tell time

// this asm is probably not really needed, but it does save about 10 instructions

// because the variables have to be volatile

asm(

"inc %[loopSteps]" "\n\t"

"brbc 1,loopend%=" "\n\t"

"inc %A[loopStepsHigh]" "\n\t"

"brbc 1,loopend%=" "\n\t"

"inc %B[loopStepsHigh]" "\n\t"

"loopend%=:" "\n\t"

:[loopSteps] "+a" (loopSteps),[loopStepsHigh] "+a" (loopStepsHigh):);

}