Building a GPSDO controller

2011 Kasper Kjeld Pedersen

kkp2011@kasperkp.dk

This is my work log on building yet another GPSDO

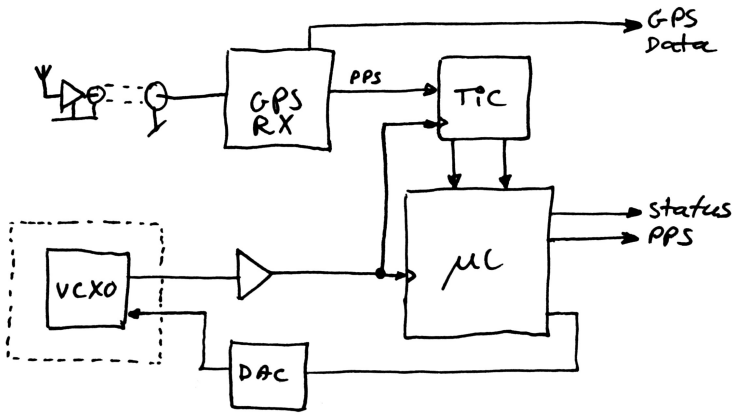
The following are done:

- Find a GPS receiver
- Verify that the GPS receiver is sound. Some are not.
- Pick a CPU and TIC design that will work with the CPU
- Test TIC performance

Todo:

- Pick a DAC and/or DAC architecture.
- Pick the OCXO. Plain OCXO, synthesized OCXO, or Rb?
- Implement a disciplining algorithm. Copy from GPSDO1?

Top level plan:



The GPS receiver delivers a time mark pulse, and we need to timestamp it against the reference oscillator. Since the cpu in the GPS and the cpu we use are likely to run at comparable clock frequencies, we need to timestamp with better than 1 clock resolution. If we do not, we get twice the error, and we get funny interference effects which are hard to impossible to suppress in software.
Once we have the timestamp, report it, and drive a control signal back to the ocxo.

The oscillator may be an OCXO, or it may be a Rubidium oscillator.

The 1ck resolution timestamp part:

My options for CPU (ie. what's currently in the drawer) is ATMega168, ATMega169, ATMega16, Xmega128, AT32UC3B0512, AT32UC3A0512, and AT91SAM9G45. The latter three have the annoying feature that the timer can not run faster than MCK/2. M168 I only have in DIP. The M169s are PV types.

The part must use a directly-fed 10-20MHz clock.
ATMega16 it is, even if it is pretty old. I forgot I had the xmega parts.

The plain capture part is easy:

If the signal is applied to the ICP1 pin, and since we need to run the timer at MCK/1, we need to extend the timer from 16 bits to a lot more. 48 bits is reasonable, as it will roll over every 162 days at 20MHz clock.

This can be done by counting overflows:

```
volatile unsigned long hi32;
ISR(SIGNAL_OVERFLOW1)
{
        ++hi32;
}
```

In the capture interrupt, hi32 may not be correct: If we capture just after the overflow, and the overflow interrupt has not fired, hi32 is one lower than it should be.

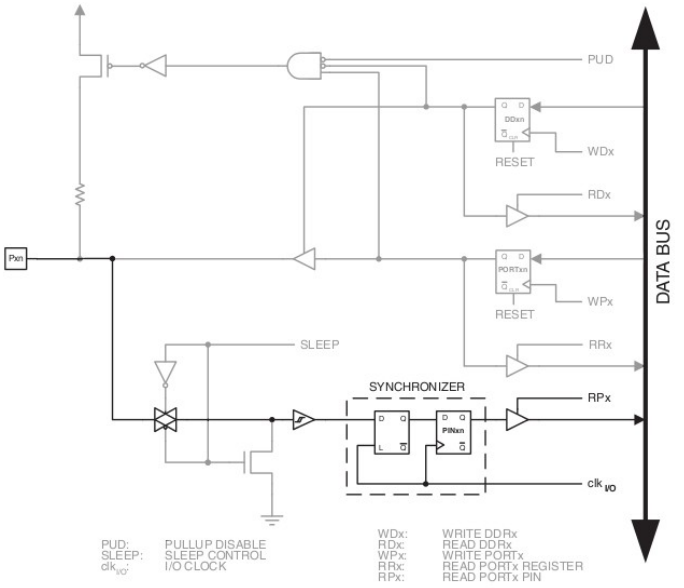
```
volatile unsigned long capture_hi32;
volatile unsigned capture_lo;
ISR(SIGNAL_CAPTURE1)
{
        unsigned n=ICP1;
        capture_lo=n;
        if ( (TIFR&(1<<OVF1)) && !(n&0x8000))
                capture_hi32=hi32+1;
        else
                capture_hi32=hi32;
}
```

This will give us a 1ck timestamp.
It seems that we can do no better than 1ck since that is the clock frequency, but in reality we can. A lot better.

And that is the fun part.

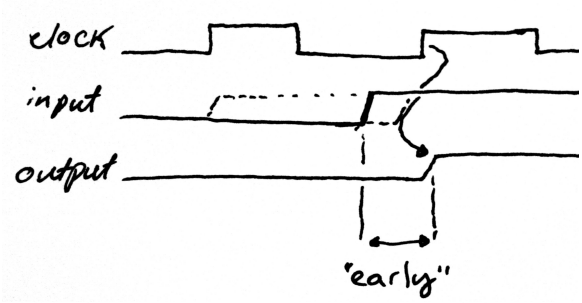
Where the AVR loses the nanoseconds, and taking them back:

When a signal arrives at the input pin of the AVR, it goes through the input schmitt trigger, and arrives at a latch. As long as the clock is high, the latch tracks the input. Then when the clock goes low, the signal is frozen, or sampled. And when the clock goes high, the frozen signal is clocked through the flipflop. This circuit is an attempt at keeping metastable states out of the core.

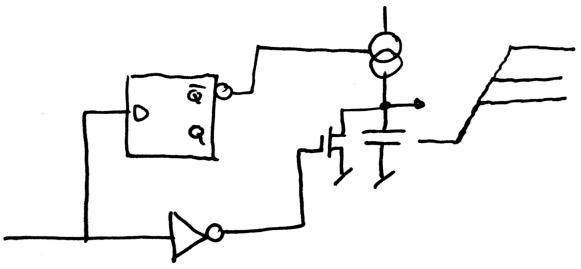


The interesting part of the above is: The signal applied to the pin will wait at the input of the latch, and the length of this delay is the difference between the signal arriving, and the clock going low. If we knew this 'early delay', which is between 0 and 1 clock, we could subtract it from the whole-clock timer value.

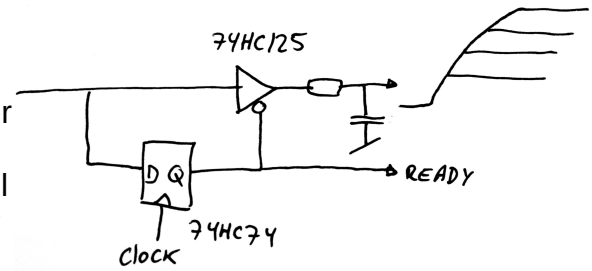
If instead of the latch-flipflop arrangement we consider a single flipflop, the delay becomes more obvious: A signal applied in a cycle will escape the flipflop at the end of the cycle.



A circuit that does this is called a Time Interpolator, or time to digital converter. There are many ways to do this, such as an integrator: Initially the capacitor is kept shorted. Then the input arrives, and the capacitor starts charging. Then when the signal passes through the flipflop, the current source is disabled, and charging stops. Properly done, 50ps resolution is easily achievable. The downside is that particularly the current source and switch require a few good components.

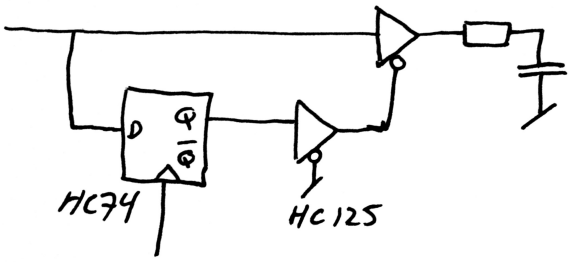


For the purposes of 'nanosecond capture', less will do the job: When the signal arrives, the tristate buffer starts charging the RC. When the signal passes through the flipflop, charging stops. The charging will be an RC curve and thus nonlinear, but this is easily correctable in software.

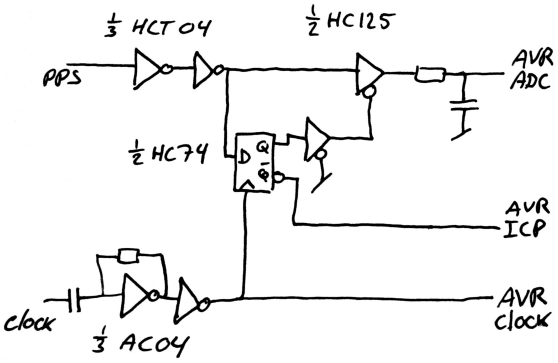


As long as the delay in the flipflop is longer than the delay in the buffer, this works. If not, the circuit will have a deadband, which is bad.

Deadband can be removed by using another buffer in the same package as a delay element. Since the buffers will be matched, now even if the flipflop is infinitely fast, the input can not tristate the final buffer before the input has had time to propagate to the output.



The complete circuit looks like this: The input signal from the GPS receiver is 3.3V, and since the TIC runs 5V, a HCT buffer is required to get to 5V. The clock is likewise external, and needs buffering before being applied to the flipflop and AVR. 200R/470p was chosen for the RC, and works well for 10MHz and up.



Timing is important: When the clock goes high, the AVR will have frozen the input 0.5 clock earlier, and will sample the input 0.5 clock in the future. Thus we have at least a +/- 25 ns (@20MHz) window around the rising edge to get the ICP signal to the AVR so as to have it recognized in the correct clock cycle.
(Taking the ICP on the opposite pin as the tristate signal attenuates any noise the AVR might inject back into the output. Since we are working with timing that is a small fraction of the rise time, all signals need to be considered as analog.)

The HC74 has a delay of 14 ns, 56% of the budget. Fortunately the AVR also contains an input clock buffer, and this also has delay. This delay becomes visible if you put one scope probe on the clock, and another on a toggling output pin. The delay of the internal clock buffer, and the output buffer, means that the external HC74 actually appears to be slightly faster than the AVR. Which is perfect.

We now have a pre-synchronized capture signal, and an analog value.
To convert this value, the ADC must already be connected to the channel connected to the capacitor, and configured to auto trigger on input capture. So ~100us after we get the input capture providing the 1ck resolution, we get an ADC interrupt telling us how early the signal arrived relative to the clock.

Since the RC charges non-linearly, first we need to convert the ADC value to linear time:

$$tp = 1024 * \ln(1.0 - (ADC/1024))$$

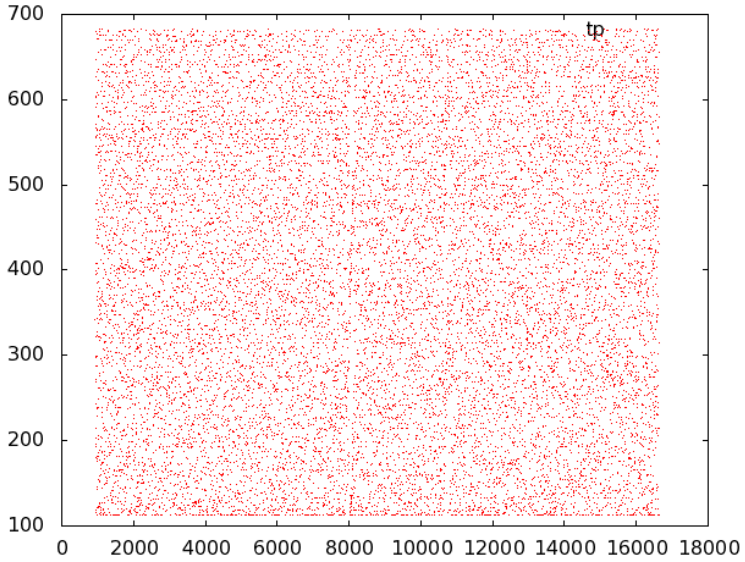
Floating point is not needed. A piecewise linear approximation is fine.
In order to scale tp to have 1 clock span, calibration is necessary. The easy way to do it is to capture a few thousand edges on a non-clock-synchronous source, and log the lowest and highest value. The difference between the high and low tp represents 1ck.

$$capturetime = (capture\_hi32 << 32) + capture\_lo - tp/(tp\_hi - tp\_lo + 1)$$

Again, we can do this without floating point.

The resulting capture, assuming an ADC span of 400 counts, is 400 times better than 1ck. With an input clock of 10MHz we should have a 250ps resolution timestamp.

Testing interpolator performance:



When you do a histogram on a large number of samples, the distribution should be flat. On the right, the samples are plotted as dots with their value, and the number. There is a higher density of samples with value 112 than other bins. That is bad.

What happens is that, when the input signal and the clock signal arrive at the inputs of the flipflop at almost the same time, there is crosstalk. So if the input is in the process of rising, and the clock signal starts to rise, it will give the input signal a small boost, and the input signal will make it through the flipflop where it otherwise would not. This leads to a nonlinearity in the time-difference to analog conversion. For the case of an AC04 providing the clock, and an HCT04 providing the input signal to a HC74, the tp span should be 2,3% larger than it is, at 14.7456MHz clock.

2.3% at 14MHz is 1.5 ns. This figure is constant, and not clock sensitive.
So we have +/- 750 ps error from signals interfering with each other in the HC74.

Before we panic, what did we get using lowly HC parts?

Clock frequency	period	improvement over capture
10MHz	100ns	*67
14.7456MHz	67.8ns	*45
16MHz	62.5ns	*42
20MHz	50ns	*33
“666 MHz”	1.5ns	*1

Performance, even with the 'bad' configuration, is equivalent to the input capture timer running at 666MHz clock.

This can be improved upon by improving the slew rate of the input signal. The HCT04 has a typical transition time of 6ns, eight times the nonlinearity. A part such as NC7SZ32 is ~10x faster, and achieves +/- 250ps.

The output noise, eyeballed from the output stream, is +/- 1 count in 256 or ~265 ps. This is the sum of the TIC noise, OCXO noise, and GPS TCXO noise.

Clocking:

To do timestamping, we need a clock.
Let us do some back of envelope calculation:

Let us say we have a 10MHz input signal, 4V p-p (+16dBm for RF people). If it is sinusoidal it can be described as

2 * sin(2*pi*10M*t)

and the derivative is

2 * 2 * pi * 10M * cos(2*pi*10M*t)

that is, the slew rate is 125MV/s, or 125V/us.

This sounds fast until we remember that we have 750ps nonlinearity in the TIC circuit. For the clock to contribute the same or less, the peak noise voltage in the clock input needs to be better than:

125MV/s * 750ps = 93mV.

Because that is how far the 10MHz clock gets in 750ps. Not very far at all.

If we only want to contribute equally to the noise figure of the TIC, the figure becomes

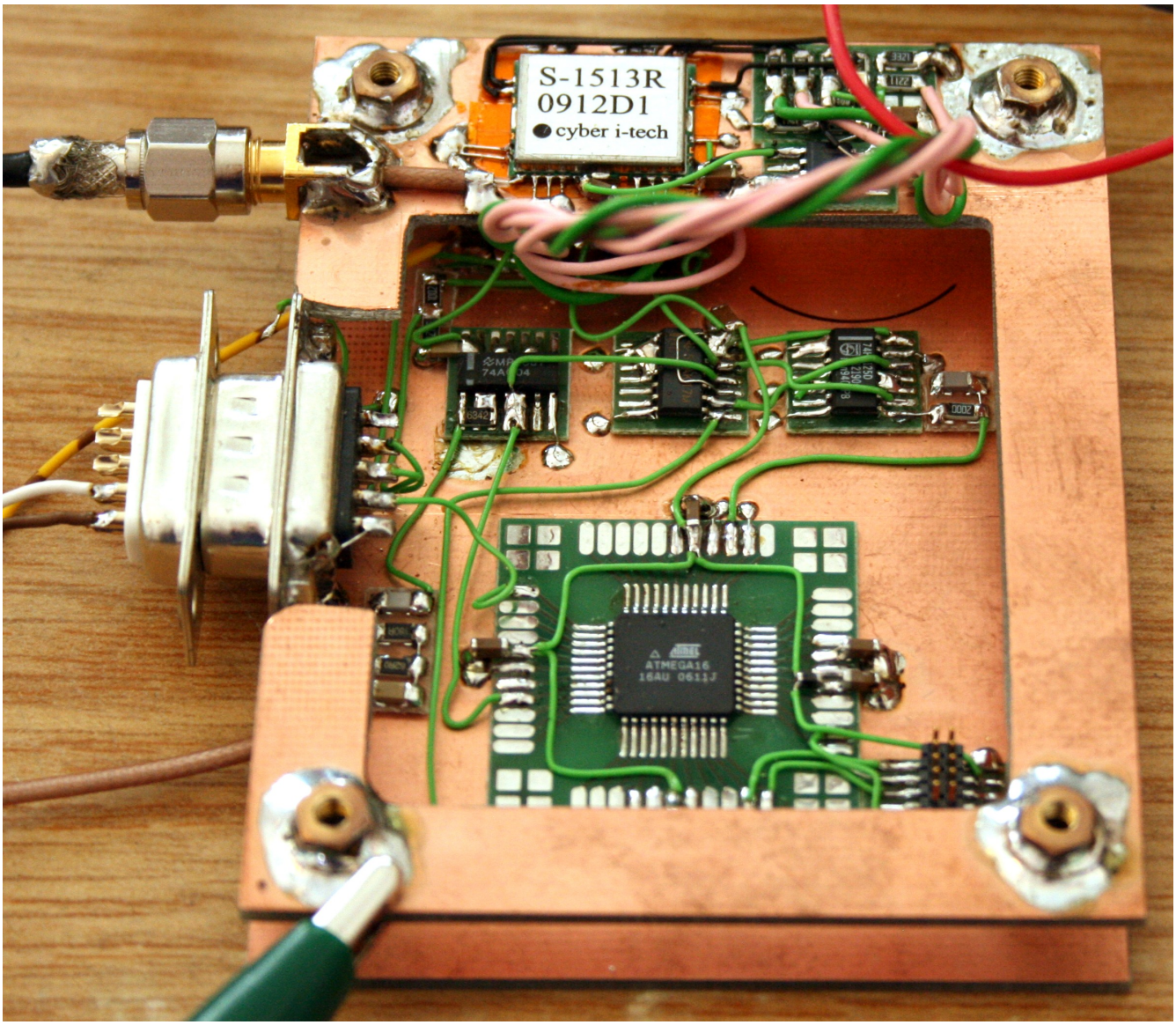
125MV/s*265ps=33mv.

This means two things:
Any part we use as a clock buffer must have clean and stable power supply.
We must have a lot less than 93mV clock crosstalk.

The internal oscillator in the AVR is worse than this. With just the core running, the noise is in this ballpark. If also driving I/O, expect +/-3ns.

Using a separate oscillator solves the problem. The same rules apply: clean stable power supply, and do not route the board in such a way that you get crosstalk.

The Device as Built:



From left to right: GPS antenna input and RF Solutions GPS-1513R. This is a ROM Venus6 receiver. It is sitting on top of two layers of Kapton tape. Next to that is stupidity protection, and 3.3V regulator.
Barely visible under the GPS module is the LM340T5 5V regulator, and the HCT04 for 3.3V/5V conversion. The next row is the AC04 clock buffer, HC74 flipflop, HC125 buffer, and RC.

The analog signal is applied to AD1, with adjacent AD0 and AD2 grounded to reduce leakage.

On the lower left is clock input termination, the OCXO feeding this board does not like operating into an open input.

The adhesive mounting boards are Wainwright MiniMount. They are no longer available, the company has been shut down.

Crazy ideas

If the clock signal is AC coupled at both ends, the capacitor should be something like
1/(2*pi*10MHz *5) = 3nF
at each end, so the line is loaded by 6nF and 25 ohm to GND.

If we couple onto the line with 2k ohm, tau becomes 12 usec. Which means that we can run 9600 bps serial over the clock line and control the OCXO that way.
300 bps is also plenty.
We have a 9600 irq. 9600/8=1200

```
//////////////////// main.c
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <string.h>

#include "suart.h"


PROGMEM unsigned ad2timet[17]= //output slew rate is 8 timestap / 1 adtick i starten
  { 0, 520, 1085, 1691, 2346, 3058, 3837, 4699, 5662, 6754, 8014, 9503, 11325, 13671, 16971, 22586, /*56783*/33000 };
int ad2t(unsigned adval) //convert ADC value to time.
{
  unsigned y0,y1,bin;
  unsigned long dy;
  bin=adval>>6;
  y0=__LPM_word(&ad2timet[bin]);
  y1=__LPM_word(&ad2timet[bin+1]);
  dy=y1-y0;
  dy*=(adval&63);
  dy+=31;
  dy>>=6;
  return y0+dy;
}



volatile unsigned long timer_high32;

volatile struct {
        unsigned      tlow; //the first 8 bytes is a 64 bit timestamp with ~ 1.5 ps resolution. better than the hardware.
    unsigned      tm16;
        unsigned long hi32;
        unsigned adc, tadc; //raw and converted time value. 400 counts on the converter @ 14.7456 => 170ps resolution
        unsigned long edge;
        unsigned char st;
} captured;



ISR(SIG_OVERFLOW1)
{
        ++timer_high32;
}

ISR(SIG_INPUT_CAPTURE1)
{
        unsigned w=ICR1;
        captured.tm16=w;
        if ((!(w&0x8000)) && (TIFR & (1<<TOV1))) {
                captured.hi32=timer_high32+1;
        } else {
                captured.hi32=timer_high32;
        }
        ++captured.edge;
        captured.st=1;

}

ISR(SIG_ADC)
{
        unsigned w=ADC & 0x03FF;
        captured.adc=w;
        captured.st|=2;
        //at this point we have captured a timestamp.
}

ISR(SIG_OUTPUT_COMPARE1A)
{

}
```

```
volatile unsigned ticl,tich;

void puthex2(unsigned char d)
{
  unsigned char k;

  k=d>>4;
  suart_put(k>9?(k+55):(k+48));
  k=d&0x0f;
  suart_put(k>9?(k+55):(k+48));}
void puthex4(unsigned d)
{
  puthex2(d>>8);
  puthex2(d);
}
void puthex8(unsigned long d)
{
  puthex4(d>>16);
  puthex4(d);
}



void main(void)
{
        unsigned long *pl;
        unsigned long oldl,oldh,newl,newh, dl,dh;

        //timer 1: max speed free running
        //ADC: input from CH1, start on input capture
        //14.7MHz input clock, /64=230400 Hz
        //thus 10 usec sample window. we want it to s/h as quick as possible.

        //interrupt on capture, interrupt on ADC complete.

        //output capture 1A runs PPS mark
        //output capture 1B runs soft usart 9600 bps = 1536 clock

  TCCR1A=0x00;
  TCCR1B=(1<<ICNC1)|1; //noice canceler, clk/1
  TIMSK |=(1<<TICIE1) | (1<<TOIE1);

  ADMUX=(1<<REFS0)+1; //5V ref, channel 1
  ADCSRA=(1<<ADEN)|(1<<ADSC)|6; //14745600/64 => 230kHz
  do {} while (ADCSRA&(1<<ADSC)); //wait for init
  SFIOR |= (1<<ADTS2)|(1<<ADTS1)|(1<<ADTS0); //ICP1 trigger
  ADCSRA|=(1<<ADIE)|(1<<ADATE); //enable interrupt, clear pending interrupt, enable auto trigger
  sei();
  asm volatile ("nop");
  asm volatile ("nop");
  asm volatile ("nop");

  initsuart();


  ticl=65535;
  tich=0;

  unsigned u;

  for (;;) {
    captured.st=0;
        do {} while (captured.st!=3);
        //calculate quantization error.
        //on the prototype with 200 ohm and 470p,
        //the total resistance is about 225 ohm and about 490pF = 107.8 ns.
        //clock is 14.7456 = 67.819 ns
        //so 1 tick is 0.629 tau.
        //slew rate is thus 1024*0.629 /clockcycle = 644 per clockcycle
        //since the adc2t function is scaled up 8, that should be 5153 counts
```

```
            //we see 4572 counts peak-peak
            //scale this to a 16 bit value. 65536/4572=14.33  65536*256/4572= 3669. that's big enough.

            u=ad2t(captured.adc);
            captured.tadc=u;
            if (ticl>u) ticl=u;
            if (tich<u) tich=u;


            unsigned long uquant;
            uquant=u; //avoid 0
            uquant*=3669;
            uquant>>=8; //scale back down. result is 24 bit and should be SUBTRACTED from timer catpture time

            //subtract quantization delay from timer capture point (quant value is higher the earlier the pulse arrives within the window)

            //subtract the low 16 bits
            captured.tlow=0xFFFF-uquant; //this can not produce a carry on the low 16 bits.
            uquant>>=16; //the high part remains and must be subtracted from the high 48 bit
            if (uquant>captured.tm16)
              --captured.hi32;
            captured.tm16-=uquant;


    asm volatile ("nop");
    asm volatile ("nop");
    asm volatile ("nop");
    asm volatile ("nop");

            suart_put('0');
            suart_put('x');
            suart_push();
            puthex8(captured.edge);
            suart_put(' ');
            suart_put('0');
            suart_put('x');
            puthex8(captured.hi32);
            puthex4(captured.tm16);
            puthex4(captured.tlow);

            suart_put(' ');
            suart_put('0');
            suart_put('x');
            puthex4(captured.tadc);
            suart_put(' ');
            suart_put('0');
            suart_put('x');
            puthex4(captured.adc);
            suart_push();

            pl=&captured;
            newl=pl[0];
            newh=pl[1];

            dl=newl-oldl;
            dh=newh-oldh;
            if (oldl>newl) --dh;
            oldh=newh;
            oldl=newl;

            suart_put(' ');
            suart_put('0');
            suart_put('x');
            puthex8(dh);
            puthex8(dl);

            suart_put(13);
            suart_put(10);
            suart_push();
  }
}
```

```
 //////////////suart.c

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>


/*
.#define SUART_RX_DIR DDRD
.#define SUART_RX_PORT PORTD
.#define SUART_RX_PIN PIND
.#define SUART_RX_BIT 2
*/

#define SUART_TX_DIR DDRB
#define SUART_TX_PORT PORTB
#define SUART_TX_BIT 5

//14756400 / 9600 = 1536    div2=768

#define HALFBITTIME 768
//#define SAMPLEEARLY 200
#define BITTIME (HALFBITTIME+HALFBITTIME);

//576-200: sample @33% tidligst.

#define RS232POL
#define POLARITY(x) (!(x))


static unsigned char su_tx_state;
static unsigned char su_tx_hold;
static unsigned char su_tx_buf[64];
static unsigned char su_tx_insert,su_tx_remove;

ISR(TIMER1_COMPB_vect)
{
  OCR1B=OCR1B+BITTIME;
  if (su_tx_state==0) {
              if (su_tx_insert!=su_tx_remove) {
                //stuff to send
#ifndef RS232POL
                SUART_TX_PORT&=~(1<<SUART_TX_BIT);
#else
                SUART_TX_PORT|=(1<<SUART_TX_BIT);
#endif
                su_tx_hold = su_tx_buf[su_tx_remove];
                if ((++su_tx_remove)>=sizeof(su_tx_buf)) su_tx_remove=0;
                su_tx_state=1;
              } else {
                //no stuff to send
                TIMSK&=~(1<<OCIE1B); //interrupt off
              }
  } else if (su_tx_state<9) { //1 2 3 4 5 6 7 8
              if POLARITY(su_tx_hold&1) SUART_TX_PORT|=(1<<SUART_TX_BIT);  //data bit
              else                      SUART_TX_PORT&=~(1<<SUART_TX_BIT);
              su_tx_hold>>=1;
              ++su_tx_state;
  } else {
#ifndef RS232POL
              SUART_TX_PORT|=(1<<SUART_TX_BIT);
#else
              SUART_TX_PORT&=~(1<<SUART_TX_BIT);
#endif
              su_tx_state=0;
  }
}


void suart_put(unsigned char d)
{
  unsigned char ins;
  ins=su_tx_insert;
```

```
  su_tx_buf[ins]=d;
  if ((++ins)>=sizeof(su_tx_buf)) ins=0;
  su_tx_insert=ins;
}

void suart_push(void)
{
  cli();
  //hvis tx interrupt er stoppet, så start den igen.
  if (!(TIMSK&(1<<OCIE1B))) {
              //output compare not enabled - not transmitting.
              OCR1B=TCNT1+BITTIME; //1 bit delay to allow stop bit
              TIMSK|=(1<<OCIE1B); //interrupt on
              TIFR=(1<<OCF1B); //clear pending irqs
  }
  sei();
}

unsigned char suart_tx_active(void)
{
  if (su_tx_insert!=su_tx_remove) suart_push();
  return TIMSK&(1<<OCIE1B);
}

unsigned char suart_tx_buffer_left(void)
{
  unsigned char i,r;
  i=su_tx_insert;
  r=su_tx_remove;
  i-=r;
  //er i under r bliver resultatet meget stort = negativt.
  if (i>sizeof(su_tx_buf)) i+=sizeof(su_tx_buf);
  //i=antal brugte bytes.
  return sizeof(su_tx_buf)-i-1;
}


void initsuart(void)
{
#ifndef RS232POL
  SUART_TX_PORT|=(1<<SUART_TX_BIT);
  SUART_TX_DIR|=(1<<SUART_TX_BIT);

#else

  SUART_TX_PORT&=~(1<<SUART_TX_BIT);
  SUART_TX_DIR|=(1<<SUART_TX_BIT);

#endif

}
```