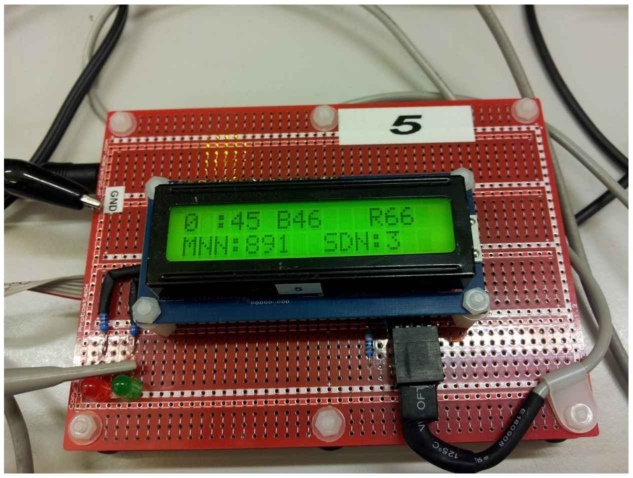
Documentation for the ECG Signals Processing System

July 7, 2015



Overview

This system uses the ATmega8 to process repeatable analogue ECG signals and computes relevant statistics. There are two output peripherals:

1. **LCD**: all relevant statistics are displayed on the LCD on a rotating basis
2. **LED**: the *green* LED flashes when a heart beat is detected

Features

- computes the following statistics in **milliseconds**:
  - average Heart Rate and NN-interval with 10^{-3} precision over 15 seconds intervals
  - standard deviations of NN-intervals and difference between adjacent NN-intervals over 5 minutes intervals
  - number and percentage of adjacent NN-intervals differing more than 50ms since system start up
- filters low frequency noise (50Hz noise tested)
- can run indefinitely

High Level Implementation

This section provides a high level overview of this system's implementation (Listing 1).

Listing 1: High Level System implementation

```
initialise hardware: Timer1, Timer0 and ADC
while (1) do
    if current ADC conversion done
        compute moving average of input
        if time < 2 seconds
            find threshold
        else
            if moving average < threshold
                compute statistics (unit == ms)
        if time changed
            display statistics
```

Design Considerations

This section outlines the key design choices made during the development of the ECG signals processing system. These choices are motivated by three major requirements: **real time performance**, **accuracy** and **hardware constraints**.

ADC

The ADC is responsible for discretising the input ECG signal in real time. In consideration for accuracy, the ATmega8's ADC is set to operate at full resolution (i.e. 10-bits); the corresponding discretisation error, ϵ, is given by Eqn 1.

$$0 \leq \epsilon < \Delta = \frac{V_{range}}{2^n - 1} = \frac{5}{2^{10} - 1} = 0.049V \tag{1}$$

This requires the ADC clock, supplied by the ATmega8 clock, to be within $[50, 200]$kHz. Since the ATmega8 provided runs at 16MHz, a prescaler of 128 is needed:

$$\frac{16MHz}{128} = 125kHz \in [50, 200]kHz$$

The ADC will operate in single conversion mode, which allows the sampling frequency to be controlled. The desired settings are enabled through the ADCSRA register (Listing 5).

Timer1 and Time Keeping

Since this system measures heart rate, precise timing is required, without which the desired statistics cannot be computed accurately. Thus Timer1 keeps a precise global time (i.e. the amount of time elapsed since system start up). In particular, Timer1 will periodically count up to EXACTLY 1 second and reset. This is done using the CTC (Clear Timer on Compare) mode with a prescaler of 1024:

$$1sec = \frac{1}{16MHz} * 1024 * cnt \implies cnt = 15625$$

EXACTLY 1 second will elapse after 15625 counts and an interrupt is generated. These settings are enabled through the TCCR1B register (Listing 3).

Listing 2: Knuth Algorithm

```
for x in ADC_data_stream:
    n = n + 1
    delta = x - mean
    mean = mean + delta / n
    M2 = M2 + delta * (x - mean)

variance = M2/(n - 1)
standard_deviation = sqrt(variance)
```

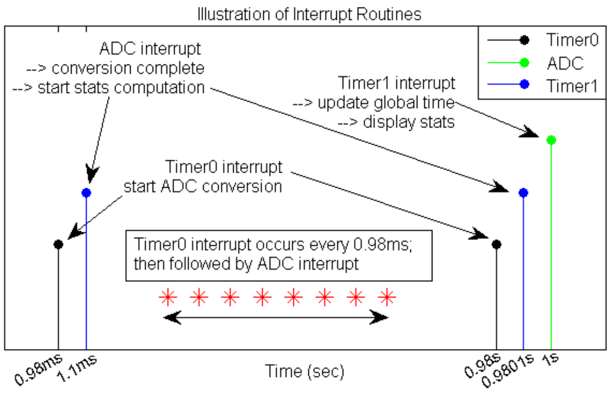


Figure 1: Timing of interrupt routines

Timer0 and Sampling Frequency

Due to ATmega8's hardware constraints, an appropriate sampling frequency, f_s, is needed so that the data processing can keep up with the ADC conversion. By the Nyquist theorem, aliasing will not occur if $f_s = 2f_{input}$. Assuming the noise in the ECG signals is up to 250Hz, $f_s = 500$Hz is adequate. However, in this system, f_s is set to 1020Hz (i.e. sampled every 0.98ms). This number is chosen as the test signals are sampled at 1kHz (allowing consistent software simulation). Timer0, instead of Timer1, starts the ADC conversions so f_s can be changed without affecting the global time keeping scheme. In addition, 1020Hz is used instead of 1kHz to avoid potential conflict with Timer1's interrupt. Listing 4 outlines Timer0 settings.

Interrupt Routines

This system has three interrupt routines: Timer1, Timer0 and ADC. Their role is to perform simple operations, i.e.: raising flags or updating volatile variables; this prevents the system from dwelling in these routines for unnecessarily long and missing other crucial events in the program. Fig 1 shows the timing of these routines.

Variable Types

Since the ATmega8 does not have a FPU, *floats* are not used as computations involving *floats* are costly, which hinders real time performance. The following data types are used:

1. **uint8_t**: 8-bit unsigned int; for flags, which are small numbers
2. **int16_t**: 16-bit signed int; for most computation results as they can handle negative numbers
3. **uint16_t**: 16-bit unsigned int; for counts as they overflow at 2^{16}
4. **uint32_t**: 32-bit signed int; for the standard deviation results to avoid quick overflow

Key Algorithms

The operation of this system relies on the robust detection of RR-intervals, real time computation of variance and mean and insusceptibility to noise. This section outlines the algorithms employed to solve these challenges.

Thresholding - Peak detection

The peak detection algorithm used in this system, illustrated in Fig 2, is a modified version of the general thresholding algorithm; the modifications are:

1. instead of detecting the R-peaks, which are more susceptible to amplitude fluctuations, S-peaks, which are more stable, are detected; as the heart signal is periodic and the S-peaks immediately follow the R-peaks, the SS-intervals are equivalent to the RR-intervals
2. as input normalisation may not be consistent, a predefined threshold may not suit all signals; thus the first 2000 samples (i.e. equivalent to 2 seconds with $f_s = 1$kHz) are used to determine an appropriate threshold; implemented in FIND_THRESHOLD

Online Variance and Mean

To display the statistics in real time and to reduce data storage requirements, the variance and mean are computed online. This is achieved with the Knuth algorithm, which is described in Listing 2.

Moving Average Filtering

One of the test signals is the superposition of a heart signal and 50Hz noise. To use the thresholding algorithm, the noise must be filtered. This is done by a moving average filter (Fig 3). Essentially, a window of length N (represented by the z^{-1} delays) moves through the input $x[n]$ and the current output $y[n]$ is the average of all values within the window (represented by the scalar b_N). This is implemented in MOVING_AVG.

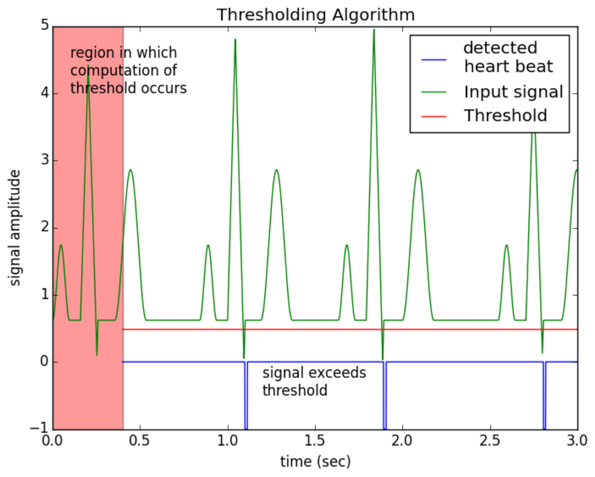
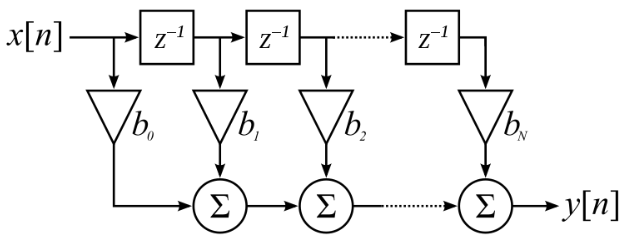


Figure 2: Illustration of the Thresholding algorithm

Figure 3: Moving Average Filter Hardware Implementation



Expected Accuracy

This section discusses the expected accuracy of the RR-detection and other statistics; considerations for ensuring best accuracy will be provided.

RR-intervals

When a beat is detected through Thresholding, the current value in Timer1's counter is recorded; the RR-intervals are given by the difference between two consecutive readings. The values from TCNT1 are converted to *ms* by dividing 16 (Eqn 2).

$$1ms = \frac{1}{16MHz} * 1024 * CNT \implies CNT \approx 16 \tag{2}$$

Thus the expected RR-interval error: $\epsilon_{RR} \leq 10^{-3}$.

Statistics Computations

Since integer division is used in computing the mean and variance (Listing 2), the loss of precision is inevitable. Thus, the means are reset every 15 seconds and the variance every 5 minutes.

Additional Considerations

The system's accuracy can be affected by the instability of the amplifier and offset unit. Since the beat detection threshold is not dynamically updated, a fluctuating gain or offset could result in erroneous beat detection. In addition, it is strongly recommended to restart the system when switching between ECG signals allowing a suitable threshold to be determined.

Appendix

Listing 3: Timer1 settings

```
// set timer 1 Mode 4, CTC on OCR1A
TCCR1B |= (1 << WGM12);
// set compare value to 15625
OCR1A = 15625;
// set interrupt on compare match
TIMSK |= (1 << OCIE1A);
// set pre-scaler to 1024
TCCR1B |= (1 << CS12) | (1 << CS10);
```

Listing 4: Timer0 settings

```
// Timer0, prescaler of 64
TCCR0 = (1 << CS01) | (1 << CS00);
// start ADC conversion every 0.98ms
TCNT0 = 10;
// enable overflow interrupt
TIMSK |= (1 << TOIE0);
```

Listing 5: ADC settings

```
ADMUX = 0; // use ADC0
ADMUX &= ~(1 << ADLAR); // 10-bit resolution
ADMUX |= (1 << REFS0); // use AVcc as reference
ADMUX &= ~( 1 << REFS1);
// prescaler = 128
ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
ADCSRA |= (1 << ADEN); // enable ADC
ADCSRA |= (1 << ADIE); // enable ADC interrupt
ADCSRA |= (1 << ADSC); // start the ADC conversion
```