The University of Aizu

BANSMOM 2010

Technical Report

Yumiko Kimezawa
2011/02/11
Adapted Systems Laboratory
Ben Abdalla Group

Contents

1 Introduction

1.1 BANSMOM

We are working on BANSMOM which is short for “smart Body Area Network System for MObility Monitoring” project. Fig. 1 shows overall view of our project. The objective of our project is to develop new telemedicine system using embedded sensors and wireless network for the elderly or the disadvantaged. In this system, they have some miniature sensors into their bodies and doctors and nurses check their health. This system makes their life comfortable.

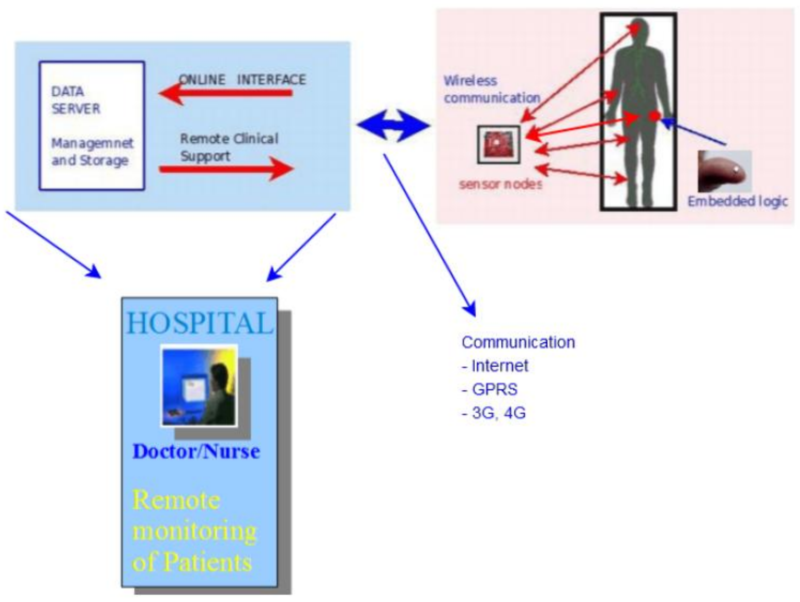

Fig. 1: Overall view of BANSMOM

1.2 Our project goal

2 Related Works

2.1 The output of an ECG records

Fig. 2 shows a typical ECG graph which is the output of an ECC recorder. The ECG graph shows time represented on the x-axis and voltage represented on the y-axis. A typical ECG tracing of the heart period consists of a P, Q, R, S, T and U wave. Each of these peaks is related to the heart activity that is essential for the medical diagnosis. Especially, the wave peaks Q, R and S is known as the QRS complex which is the most significant wave in the ECG graph. The QRS complex is used in many methods of the ECG analysis.

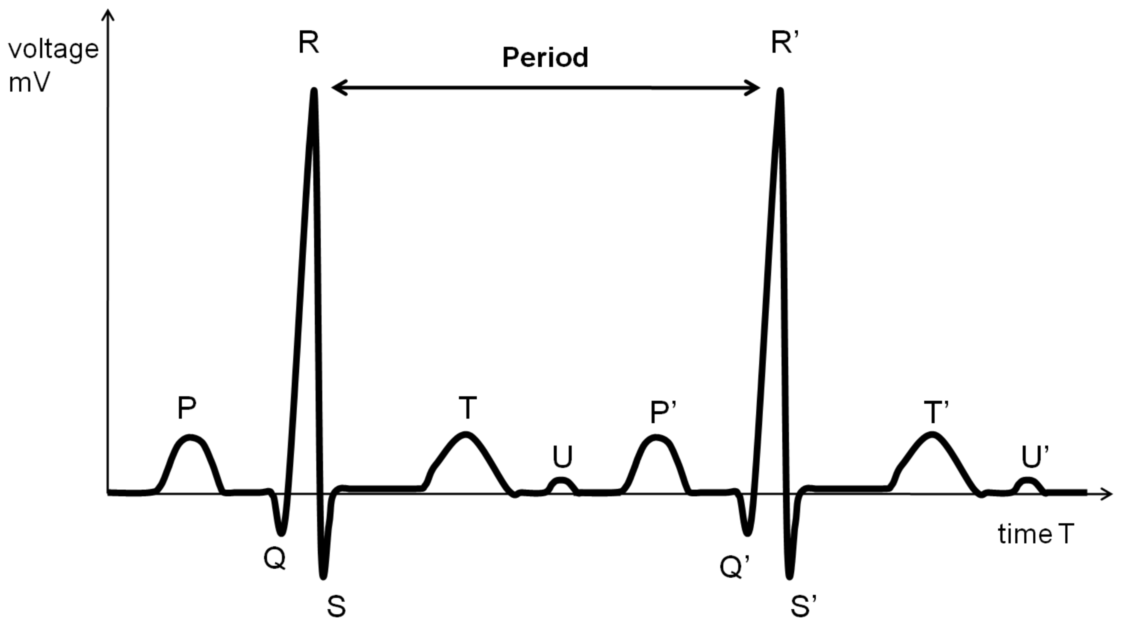

Fig. 2: A typical ECG Graph

2.2 12-lead ECG

A 12-lead ECG, a simplified method, is that twelve different ECG signals are recorded at approximately the same time from six sensors put on the patient skin keeping the patient at rest. The time required to put leads on the patient skin is only a few minutes. Therefore, it is simple and does not put pressure on patients. However, as it outputs only waveforms, it is impossible to diagnose whether the patient's heart is normal or not. When the patient has problems such as heart palpitation or shortness of breath, 12-lead ECG does not provide data correctly.

2.3 Holter monitor

To overcome the shortcomings of 12-lead ECGs, Holter monitors are used instead. In the examination using a Holter monitor, ECG signals are collected from the patient over a day. With this examination, it is possible to detect temporary irregular heartbeat undiagnosed with 12-lead ECG.

3 Period-Peaks Detection (PPD) Algorithm

Period-Peaks Detection (PPD) algorithm was used to analyze ECG signals. Heart period, typical peaks (P, Q, R, S, T and U) and time spans of inter-peak (R-R interval) were calculated by using the algorithm. The algorithm is based on Pan-Tompskins algorithm. Fig. 3 shows the processing flow of PPD algorithm. The algorithm consists of two execution flows: First flow finds the heart period using the autocorrelation function and second flow finds the number, amplitude and time interval of time interval of the peaks. This algorithm detects the heart period first and then looks for all typical peaks. ECG signals contain many irregular peaks. This methodology prevents irregular peaks from being detected as typical peaks. Fig. 4 shows finding intervals algorithm. Fig. 5 shows finding peaks algorithm. These two algorithms are the backbone of our PPD algorithm.

3.1.1 Reading data

In this function, the filtered ECG data is readied for using PPD algorithm from the buffer.

3.1.2 Derivation

In this function, the ECG signal is differentiated to increase signal peaks. The derivative formula used by this algorithm is (1). This formula has only operation of subtraction. Therefore this function won't have to consume a lot of arithmetic operations, which is multiplication that may tire the system in time and power and a very helpful step for this algorithm.

$$\frac{\partial \mathbf{y}}{\partial \mathbf{t}}(\mathbf{t}) \approx \frac{\mathbf{y}[\mathbf{n}+\mathbf{1}]-\mathbf{y}[\mathbf{n}]}{(\mathbf{n}+\mathbf{1})-\mathbf{n}} = \mathbf{y}[\mathbf{n}+\mathbf{1}]-\mathbf{y}[\mathbf{n}] \quad (\mathbf{1})$$

With this derivative, when there is a peak it will be increased with relative to the samples before it, and if the value of $y[n]$ and $y[n+1]$ are near to each other (i.e. no peaks) then the difference will look relatively smaller on the new derivative graph. The advantage of taking the derivative, and thus adding some overhead to the code, is that the fluctuations taking place in the signal and especially those around the peaks would be reduced to a near-zero-value. In addition, performance overhead associated with derivative calculation of the ECG signal is negligible compared to the rest of the algorithm.

3.1.3 Autocorrelation

In this function, periodicity of the ECG signal is found by using autocorrelation function (ACF). The ACF shown in (2) is a statistical method used to measure the degree of association between values in a single series separated by some lags. The fixed length ACF is defined by (3).

$$R_y[k] = \sum_{n=-\infty}^{n=\infty} y[n] \times y[n-k] \qquad (2)$$

$$R_y[L] = \sum_{n=0}^{N} y[n] \times y[n-L] \qquad (3)$$

R_y is the autocorrelation function, $y[n]$ is the ECG filtered signal, and L is a positive natural number related to the number of times needed for lags of the calculations to get the period, same as the number of lags of the autocorrelation.

3.1.4 Finding interval

In this function, interval points in ECG signal based on results of previous function.

3.1.5 Extraction

In this function, significant peaks from calculated interval information flow are extracted.

3.1.6 Discrimination

In this function, traditional six peak points, P, Q, R, S, T and U, from extracted peaks.

3.1.7 Store results

In this function, intervals and peaks of information calculated in each of the flows are stored in buffer.

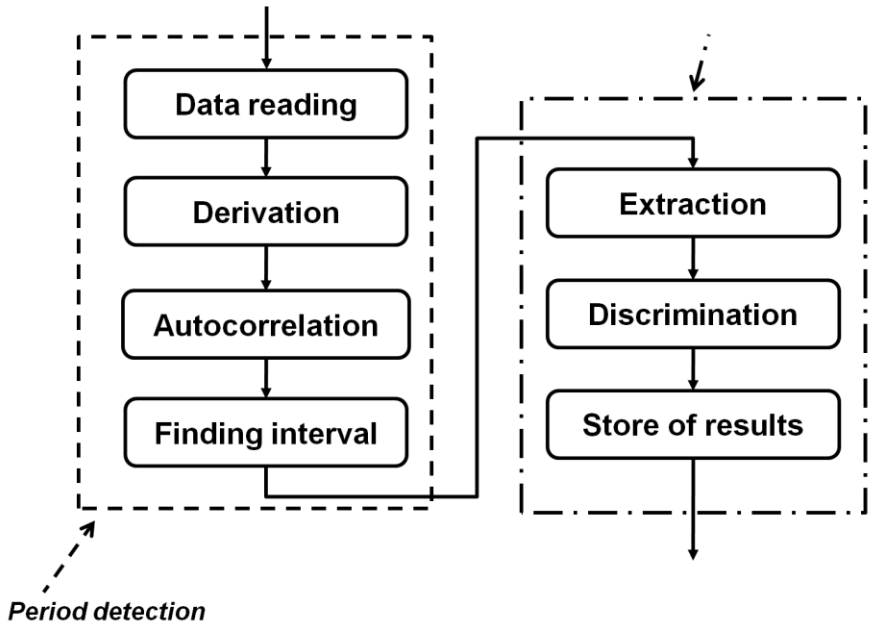

Fig. 3: The Processing Flow of PPD Algorithm

3.2 Period detection flow

Fig. 4 shows finding intervals algorithm. This algorithm contains 7 steps, finding maximum value, reducing negative value, detecting peaks from ACF results, finding basing points, sort of basing points, calculating the interval and renewing next start index. First, in the finding maximum value step, a maximum value and its location are found. By finding the maximum value, a threshold is determined because it is based on the maximum value. Second, in the reducing negative value step, when input value is negative, the value is replaced by 0. Third, in the detecting peaks from ACF results step, all peaks are detected and counted and their location are found. Forth, in the finding basing points step, high threshold and low threshold are determined. Fifth, in the sort basing points, base points are arranged in ascending order. Sixth, in the calculating intervals, intervals are found. Finally, in the renewing next start index, start index is renewed.

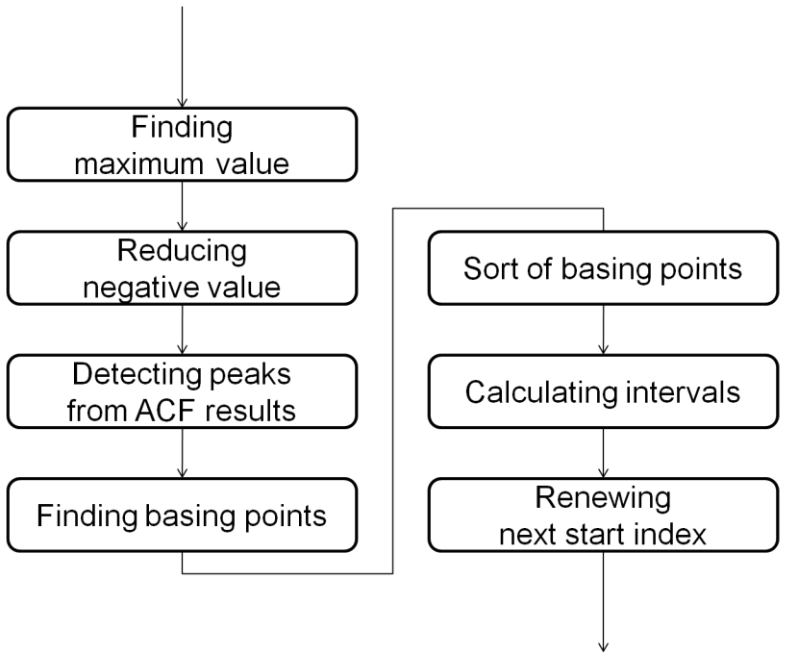

Fig. 4: The Processing Flow of Period Detection Process

3.3 Peaks processing flow

Fig. 5 shows finding peaks algorithm. First, in the finding positive peaks step, the number of positive peaks is counted and their addresses are stored. Second, in the finding negative peaks step, the number of negative peaks is counted and their addressed are stored. It is easier to find each peak by dividing all peaks into two groups positive and negative. Third, in the finding R peak step, the maximum value is found from among positive peaks in an interval. The peak is identified as the R peak. Fourth, in the finding P peak step, the maximum value is found from among positive peaks from the start to maximum value index. The peak is identified as the P peak. Fifth, in the T peak step, the maximum value is found from among positive peaks from maximum value + 1 to end. The peak is identified as the T peak. Sixth, in the finding Q peak step, the negative peak closest to the R peak is found from among negative peaks from start to R peak point. The peak is identified as the Q peak. Finally, in the finding S peak step, the negative peak closest to the R peak is found from among negative peaks from R peak point to end.

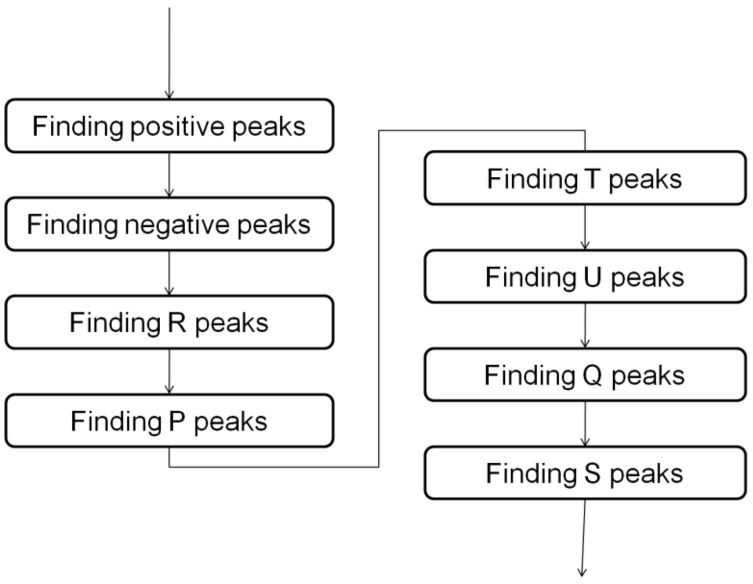

Fig. 5: The Processing Flow of Peaks Processing process

4 System Architecture

Fig. 6 shows the desired system architecture. This system consists of four phases, Signal reading, Filtering, Analysis and Display.

4.1.1 Signal reading

10 tiny sensors will be used to sense ECG signals. The number of the read data from the sensor is extensible 15 or more. The size of data is included in the read data as well as data from the sensor.

4.1.2 Filtering

In this phase, the bandpass filter based on Finite Impulse Response (FIR) filter. The bandpass filter is cascaded the low-pass and high-pass filters, is used. This filter reduces the influence of muscle noise, 50 Hz interference, baseline wander and T-wave interference.

4.1.3 Analysis

This phase read filtered data from the internal buffer. This data is analyzed by using a PPD algorithm. Analyzed data is stored in the internal buffer.

4.1.4 Display

External monitor outputs the results of analysis. The output data is peaks of each typical wave (P, Q, R, S, T and U wave), heart rate and entire waveform. While running the system, the external monitor outputs the results at real time.

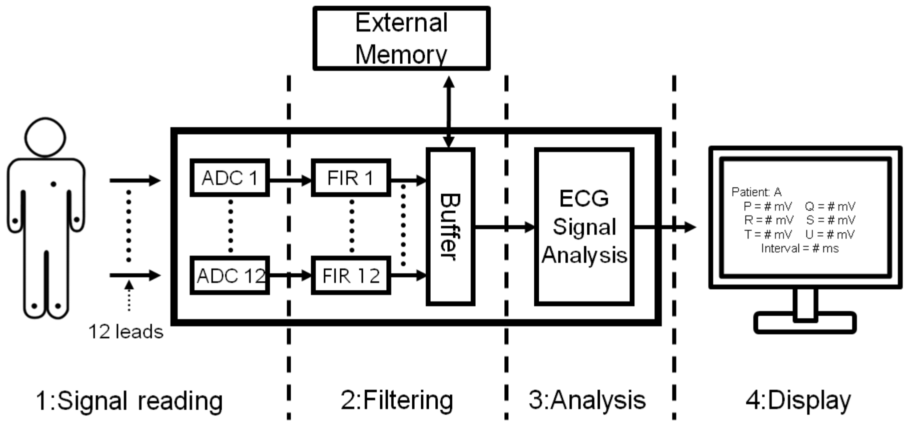

Fig. 6: The Desired System Architecture

5 Methods

Fig. 7 shows the system block diagram of our prototype system. Altera Quartus II, Altera SOPC Builder and AlteraMegaCore Function were used to design a prototype system and Altera Nios II IDE was used to develop PPD algorithm that runs on the prototype system. Our prototype system includes one Master module and several PPD modules. The master module consists of Altera Nios II processor, four on-chip memories that used for storing raw ECG data, processor memory, shared memory and memory simulated as external memory, several FIR filters, graphic LCD controller, LED controller and JTAG UART which is used for connecting to a host PC. One PPD module consists of Altera Nios II processor, on-chip memory and interrupt timer. Each of these components is connected by Altera Avalon Bus. FIR filter is generated by Altera MegaCore Function. Specifications of the filter are: filter steps are 51, a sample rate is 128 and a cutoff frequency is from 5 Hz to 15 Hz. The sample data actually collected from the patients are used for analysis of an algorithm. The sample data actually collected from the patients are used for analysis of an algorithm from the database of MIT-BIH Normal Sinus Rhythm. The length of the sample data is ten seconds. We have made a minor change in a prototype system proposed before for parallel processing of ECG signals from several leads. I design some systems allow parallel processing of ECG signals. As evaluation of systems, we prepare some sample data from the database. We compare the execution time between each system.

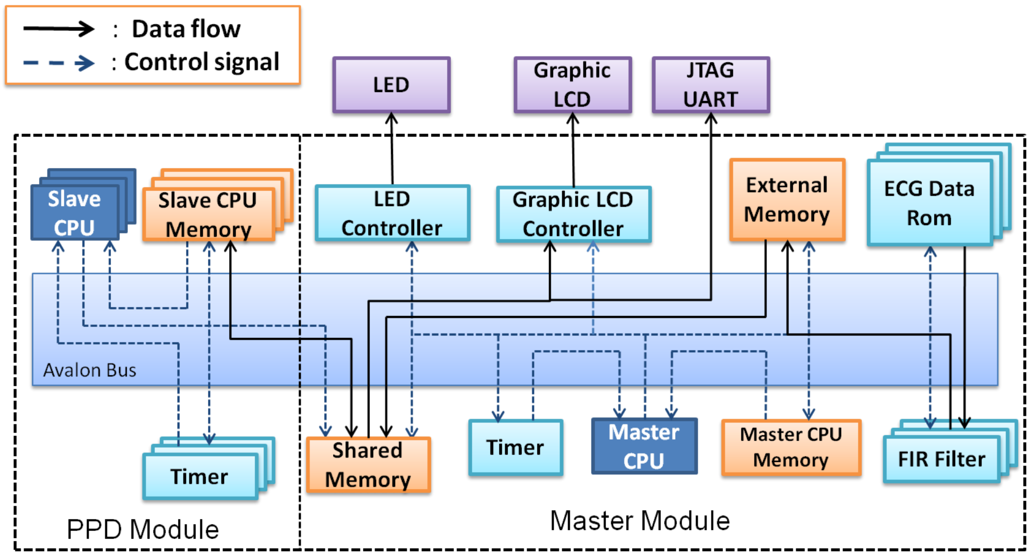

Fig. 7: The System Block Diagram

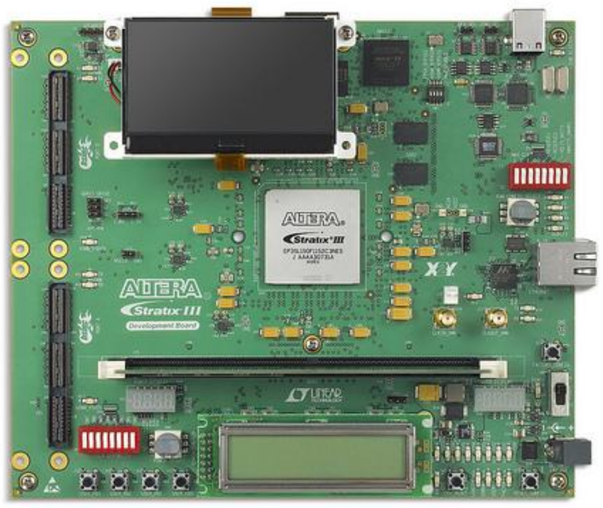

Fig. 8: Stratix III DSP bord

6 Evaluation

Prototype systems, 2-lead and 3-lead were designed for this research. Table 1 shows the results of logic synthesis of three systems. In the 1-lead system, logic utilization was about 15%. Block memory bits made up about 22%of the total. The maximum operating frequency was 96.43MHz. Power was 621.53mW. In the 2-lead system, logic utilization was about 27%. Block memory bits made up about 42% of the total. The maximum operating frequency was 132.49MHz. Power was 627.23mW. In the 3-lead system, logic utilization was about 38%. Block memory bits made up about 52% of the total. The maximum operating frequency was 119.42MHz. Power was 632.41mW. From their results, the scale of systems seemed to be larger in the number of leads because the number of modules was increased with an increase of leads. Twelve sample data were used in evaluation of each system. In the 1-lead system, one sample data was processed twelve times serially. In the 2-lead system, two sample data were processed six times serially. In the 3-lead system, three sample data were processed four times serially. Table 2 shows the individual execution time of processing and total time to process twelve sample data of each system. Record No. shows that which sample data was used from the database. In the 1-lead system, it took 134.504 seconds to process twelve sample data. In the 2-lead system, it took 83.138 seconds and in the 3-lead system, it took 67.899 seconds. Comparing the total time spent on processing of twelve sample data by using the 1-lead system, total time was decreased about 38% when the 2-lead system was used and decreased about 50% when the 3-lead system was used.

7 Conclusion

The system designed before was able to process the ECG signals only at the same time and it took longer to process the ECG signals from many leads. Therefore, the system designed before had to be improved for parallel processing of the ECG signals from some leads. Compared with the 1-lead system, execution time in the 2-lead system was decreased about 38% and execution in the 3-lead system was decreased about 50%. In this research, parallel processing of the ECG signals from several leads became possible and execution time to process the ECG signals from several leads reduced about 50%. The problems of this research may be the large scale of systems and accuracy of PPD algorithm as mentioned before. These problems have to be solved. The system can process the ECG signals from twelve or more leads had not developed yet. Although the system had been proposed in this research, it was difficult to develop that system because one core needed per one PPD module and area was too large. In the future,

8 Future Work

PPD modules will be divided into some tasks and these tasks will be mapped to many cores. PPD algorithm will be optimized by using OpenMP or MPI to parallelize the C code and test it in the multicore system.

9 Appendix

9.1 Source code of PPD algorithm

9.1.1 Reading data function

```
int reading_data( __int64* _read_values, int* _current_start_index ){
    alt_mutex_dev *mutex_hdl;
    int i;
    int status;
    int get_data;
    int function_flag;

    // get device handle
    mutex_hdl = altera_avalon_mutex_open( ONCHIP_SHARED_BUFFER_MUTEX_NAME );
    status = 0;
    get_data = 0;
    function_flag = FUNCTION_FAILURE;
    // shared device try lock
    if( altera_avalon_mutex_trylock( mutex_hdl, 1 ) == 0 ){
        status = SHARED_MEM_STATUS_RD;
        // data get from Shared Memory
        if( status == PPD_READY ){
             // get filtered data
            for( i = 0; i < DEV_STEP; i++ ){
                _read_values[i] = SHARED_MEM_RD( SM_FILTERED_DATA_BASE+i );
            }
             // get current start index
            *_current_start_index = SHARED_MEM_RD( SM_CURRENT_START_INDEX_BASE );
             // status renew
            SHARED_MEM_STATUS_WR( (status&PPD_FINISH_MSK) | PPD_FLG_CLR );
```

```
                function_flag = FUNCTION_TRUE;
            }
            else{
                function_flag = FUNCTION_FAILURE;
            }
            // shared `device unlock
            altera_avalon_mutex_unlock( mutex_hdl );
            return function_flag;
        }
        else{
            return FUNCTION_FAILURE;
        }
}
```

9.1.2 Derivation function

```
void derivation( const __int64* _input_values, __int64* _output_values ){
   int i;

   for( i = 0; i < ACF_STEP; i++ ){
      _output_values[i] = _input_values[i+1] - _input_values[i];
   }
}
```

9.1.3 Autocorrelation function

```
void autocorrelation( const __int64* _input_values, __int64* _output_values ){
   int i, j;
   __int64 result;

   /** initialize **/
   result = 0;

   /** running **/
   for( j = 0; j < ACF_STEP; j++ ){
      result = 0;
      // calculate
      for( i = 0; i < ACF_STEP; i++ ){
          if( i+j < ACF_STEP ){
             result = result + ( _input_values[i] * _input_values[i+j] );
          }
          else{
             result = result + 0;
          }
       }
       // store result
       _output_values[j] = result;
     }
}
```

9.1.4 Find interval function

```
// find maximum value
   max_value = _input_values[0];
   max_value_index = 0;
   for( i = 1; i < ACF_STEP; i++ ){
      if( _input_values[i] > max_value ){
         max_value = _input_values[i];
         max_value_index = i;
      }
   }
   // reduce negative value
   for( i = 0; i < ACF_STEP; i++ ){
      reduced_negative_values[i] = ( _input_values[i] < 0 ) ? 0 : _input_values[i];
   }
   reduced_negative_values[ACF_STEP] = 0;
   // peak detection from ACF result
   previous_sign = POSITIVE;
   current_sign = POSITIVE;
   peak_points[0] = 1;
   peak_points[peak_points[0]] = max_value_index;
   for( i = 0; i < ACF_STEP; i++ ){
      current_sign = ( reduced_negative_values[i]-reduced_negative_values[i+1] < 0 ) ?
NEGATIVE : POSITIVE;
```

```
        if( (previous_sign == NEGATIVE) && (current_sign == POSITIVE) && (max_value_index !=
i)){
            peak_points[0] = peak_points[0] + 1;
            peak_points[peak_points[0]] = i;
        }
        previous_sign = current_sign;
    }
    // find basing points
    threshold_high = max_value * ACF_PEAK_DETECTION_THRESHOLD_RATE_HIGTH;
    threshold_low = max_value * ACF_PEAK_DETECTION_THRESHOLD_RATE_LOW;
    basing_points[0] = 0;
    for( i = 1; i <= peak_points[0]; i++ ){
        if( 0 <= peak_points[i] && peak_points[i] < ACF_PEAK_DETECTION_THRESHOLD_AREA_HIGTH ){
            if( _input_values[peak_points[i]] > threshold_high ){
                basing_points[0] = basing_points[0] + 1;
                basing_points[basing_points[0]] = peak_points[i];
            }
        }
        else if( ACF_PEAK_DETECTION_THRESHOLD_AREA_HIGTH <= peak_points[i] && peak_points[i]
< ACF_PEAK_DETECTION_THRESHOLD_AREA_LOW ){
            if( _input_values[peak_points[i]] > threshold_low ){
                basing_points[0] = basing_points[0] + 1;
                basing_points[basing_points[0]] = peak_points[i];
            }
        }
    }
    // sort basing points
    for( i = 1; i < basing_points[0]; i++ ){
        for( j = i+1; j < basing_points[0]; j++ ){
            if( basing_points[i] > basing_points[j] ){
                tmp_basing_point = basing_points[i];
                basing_points[i] = basing_points[j];
                basing_points[j] = tmp_basing_point;
            }
        }
    }
    // calculate interval
    interval_sum = 0;
    *_interval_info_num = 0;
    if( basing_points[0] > 1 ){
        for( i = 1; i < basing_points[0]; i++ ){
            (_interval_info+*_interval_info_num)->start_index =
basing_points[i];//(_next_start_index-ACF_STEP) + basing_points[i];
            (_interval_info+*_interval_info_num)->end_index =
basing_points[i+1];//(_next_start_index-ACF_STEP) + basing_points[i+1];
            (_interval_info+*_interval_info_num)->interval = basing_points[i+1] -
basing_points[i];
            interval_sum = interval_sum + (_interval_info+*_interval_info_num)->interval;
            *_interval_info_num = *_interval_info_num + 1;
        }
    }
    // renew next start index
    if( interval_sum != 0 ){
        *_next_start_index = _current_start_index + ( interval_sum + basing_points[1] );
    }
    else{
        *_next_start_index = _current_start_index + ACF_STEP;
    }
}
```

9.1.5 Peaks detection function

```
void peaks_detection( const __int64 *_input_data, const int _interval_num, const
interval_information *_interval_info, peaks_information *_peaks_info ){
    int i, j;
    __int64 max_value;
    int max_value_index, p_wave_index, t_wave_index;
    char current_sign, previous_sign;
    int peak_points_positive[ECG_PEAK_DETECTION_MAX],
peak_points_negative[ECG_PEAK_DETECTION_MAX];
    peaks_information peak_point;
```

```c
peaks_information_64 peak_value;
#ifdef __DEBUG_MODE__
   time_information_us_int start_time, end_time, tmp_time;
#endif

for( i = 0; i < _interval_num; i++ ){
   #ifdef __DEBUG_MODE__
      start_time = g_exe_time.elapsed;
   #endif

   peak_point.p = PEAKS_DETECTION_FAILURE;
   peak_point.q = PEAKS_DETECTION_FAILURE;
   peak_point.r = PEAKS_DETECTION_FAILURE;
   peak_point.s = PEAKS_DETECTION_FAILURE;
   peak_point.t = PEAKS_DETECTION_FAILURE;
   peak_point.u = PEAKS_DETECTION_FAILURE;
   peak_value.p = 0;
   peak_value.q = 0;
   peak_value.r = 0;
   peak_value.s = 0;
   peak_value.t = 0;
   peak_value.u = 0;
   t_wave_index = 0;
   // find positive peaks
   current_sign = POSITIVE;
   previous_sign = POSITIVE;
   peak_points_positive[0] = 0;
   for( j = (_interval_info+i)->start_index; j < (_interval_info+i)->end_index; j++ ){
      current_sign = ( _input_data[j]-_input_data[j+1] < 0 ) ? NEGATIVE : POSITIVE;
      if( (previous_sign == NEGATIVE) && (current_sign == POSITIVE) ){
         peak_points_positive[0] = peak_points_positive[0] + 1;
         peak_points_positive[peak_points_positive[0]] = j;
      }
      previous_sign = current_sign;
   }
```

```
// find negative peaks
current_sign = NEGATIVE;
previous_sign = NEGATIVE;
peak_points_negative[0] = 0;
for( j = (_interval_info+i)->start_index; j < (_interval_info+i)->end_index; j++ ){
   current_sign = ( _input_data[j]-_input_data[j+1] < 0 ) ? NEGATIVE : POSITIVE;
   if( (previous_sign == POSITIVE) && (current_sign == NEGATIVE) ){
      peak_points_negative[0] = peak_points_negative[0] + 1;
      peak_points_negative[peak_points_negative[0]] = j;
   }
   previous_sign = current_sign;
}
if( peak_points_positive[0] == 0 || peak_points_negative[0] == 0 ){
   //peaks detection failure
   break;
}

#ifdef __DEBUG_MODE__
   end_time = g_exe_time.elapsed;
   tmp_time = get_execution_time( start_time, end_time );
   g_exe_time.extra = exe_time_add( g_exe_time.extra, tmp_time );
   start_time = g_exe_time.elapsed;
#endif

// find R peak
max_value = _input_data[peak_points_positive[1]];
max_value_index = 1;
for( j = 2; j <= peak_points_positive[0]; j++ ){
   if( max_value < _input_data[peak_points_positive[j]] ){
      max_value = _input_data[peak_points_positive[j]];
      max_value_index = j;
   }
}
peak_point.r = peak_points_positive[max_value_index];
peak_value.r = max_value;
```

```
// find P peak
peak_point.p = peak_points_positive[1];
peak_value.p = _input_data[peak_point.p];
p_wave_index = 1;
for( j = 2; j < max_value_index; j++ ){
    if( peak_value.p < _input_data[peak_points_positive[j]] ){
        peak_point.p = peak_points_positive[j];
        peak_value.p = _input_data[peak_point.p];
        p_wave_index = j;
    }
}
// find T peak
if( max_value_index+1 <= peak_points_positive[0] ){
    for( j = max_value_index+1; j <= peak_points_positive[0]; j++ ){
        if( peak_value.t < _input_data[peak_points_positive[j]] ){
            peak_point.t = peak_points_positive[j];
            peak_value.t = _input_data[peak_point.t];
            t_wave_index = j;
        }
    }
}
// fine U peak
if( t_wave_index+1 <= peak_points_positive[0]){
    for( j = t_wave_index+1; j <= peak_points_positive[0]; j++ ){
        if( peak_value.u < _input_data[peak_points_positive[j]] ){
            peak_point.u = peak_points_positive[j];
            peak_value.u = _input_data[peak_point.u];
        }
    }
}
else if( 1 < p_wave_index ){
    for( j = 1; j < p_wave_index; j++ ){
        if( peak_value.u < _input_data[peak_points_positive[j]] ){
            peak_point.u = peak_points_positive[j];
            peak_value.u = _input_data[peak_point.u];
```

```
                }
            }
        }
        // find Q peak
        for( j = 1; j <= peak_points_negative[0]; j++ ){
            if( peak_points_negative[j] < peak_point.r ){
                peak_point.q = peak_points_negative[j];
                peak_value.q = _input_data[peak_points_negative[j]];
            }
        }
        // find S peak
        for( j = 1; j <= peak_points_negative[0]; j++ ){
            if( peak_point.r < peak_points_negative[j] ){
                peak_point.s = peak_points_negative[j];
                peak_value.s = _input_data[peak_points_negative[j]];
                break;
            }
        }
        // store results
        (_peaks_info+i)->p = peak_point.p;
        (_peaks_info+i)->q = peak_point.q;
        (_peaks_info+i)->r = peak_point.r;
        (_peaks_info+i)->s = peak_point.s;
        (_peaks_info+i)->t = peak_point.t;
        (_peaks_info+i)->u = peak_point.u;

        #ifdef __DEBUG_MODE__
            end_time = g_exe_time.elapsed;
            tmp_time = get_execution_time( start_time, end_time );
            g_exe_time.discrim = exe_time_add( g_exe_time.discrim, tmp_time );
        #endif
    }
}
```

9.1.6 Store results function

```
void store_results( const int _next_start_index, const int _interval_info_num, const
interval_information* const _interval_info, const peaks_information* const _peaks_info ){
   alt_mutex_dev *mutex_hdl;
   int i;
   int base_addr;
   int status;
   #ifdef __DEBUG_MODE__
      time_information_us_int start_time, end_time, tmp_time;
      start_time = g_exe_time.elapsed;
   #endif

   // get device handle
   mutex_hdl = altera_avalon_mutex_open( ONCHIP_SHARED_BUFFER_MUTEX_NAME );

   // shared device lock
   altera_avalon_mutex_lock( mutex_hdl, 1 );
   /* store results to shared memory */
   // next start index
   SHARED_MEM_WR( SM_NEXT_START_INDEX_BASE, _next_start_index );
   // # of interval
   SHARED_MEM_WR( SM_NUM_OF_INTERVAL_BASE, _interval_info_num );
   // PPD results
   base_addr = SM_PPD_RESULT_BASE;
   for( i = 0; i < _interval_info_num; i++ ){
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_INTERVAL_START,
_interval_info[i].start_index );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_INTERVAL_END, _interval_info[i].end_index );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_INTERVAL_VAL, _interval_info[i].interval );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_P_POINT, _peaks_info[i].p );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_Q_POINT, _peaks_info[i].q );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_R_POINT, _peaks_info[i].r );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_S_POINT, _peaks_info[i].s );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_T_POINT, _peaks_info[i].t );
      SHARED_MEM_WR( base_addr+SM_PPD_RESULT_U_POINT, _peaks_info[i].u );
```

```
        base_addr = base_addr+SM_PPD_RESULT_OFFSET;
    }
    // status renew
    status = SHARED_MEM_STATUS_RD;
    SHARED_MEM_STATUS_WR( (status&PPD_FINISH_MSK) | PPD_FINISH );

    #ifdef __DEBUG_MODE__
        end_time = g_exe_time.elapsed;
        tmp_time = get_execution_time( start_time, end_time );
        g_exe_time.store = exe_time_add( g_exe_time.store, tmp_time );

        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x0, g_exe_time.reading.ms );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x1, g_exe_time.reading.s );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x2, g_exe_time.div.ms );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x3, g_exe_time.div.s );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x4, g_exe_time.acf.ms );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x5, g_exe_time.acf.s );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x6, g_exe_time.find.ms );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x7, g_exe_time.find.s );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x8, g_exe_time.extra.ms );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0x9, g_exe_time.extra.s );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0xa, g_exe_time.discrim.ms );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0xb, g_exe_time.discrim.s );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0xc, g_exe_time.store.ms );
        SHARED_MEM_WR( EXE_TIME_COUNTERS_BASE+0xd, g_exe_time.store.s );
    #endif

    // shared device unlock
    altera_avalon_mutex_unlock( mutex_hdl );
}
```

10 References