IMPLEMENTATION OF A REAL-TIME QRS DETECTOR WITH A GRAPHICAL PROGRAMMING LANGUAGE

Monnie Anderson

National Instruments Corp.
Austin, Texas

ABSTRACT

This article describes in detail how to implement a complete data acquisition, analysis, and display program in LabVIEW. The program measures an ECG signal, detects the QRS complexes, and displays the results in real time. LabVIEW is a graphical, block diagram programming language for the Macintosh computer that is designed primarily for scientists and engineers, who typically employ block diagrams to describe or design applications. Many of these users find traditional languages difficult to use or do not program with them often enough to become or remain proficient. For readers unfamiliar with the language, the article contains explanations of fundamental LabVIEW concepts and terminology.

BACKGROUND

Several biomedical users of LabVIEW have asked how to use it to measure instantaneous heart rate. Heart rate monitors or cardiotachometers usually measure an average rate. For those instruments that do measure instantaneous rate, their measurements are not easily time-aligned to other digitized physiological signals. An alternative approach is to sample the ECG signal and then to use software algorithms to locate QRS peaks in the digitized signal and to compute the instantaneous RR interval rate from the peak positions. The result is as accurate as the sample interval used.

Jiapu Pan and Willis J. Tompkins of the University of Wisconsin have developed a real-time QRS detection algorithm [1] for a Z-80 microprocessor. Patrick S. Hamilton and Willis Tompkins have implemented a version of this algorithm in C to run on an IBM PC. [2] This implementation demonstrated a sensitivity of 99.69% and a positive predictivity of 99.77% when tested against the MIT/BIH database. At the suggestion of Willis Tompkins, I refer to these algorithms collectively as the **Wisconsin QRS detector**, or more simply, the Wisconsin algorithm.

The Wisconsin algorithm first filters the measured ECG signal to remove noise and wander and to accentuate QRS complexes relative to other events. It then detects candidate peaks in the filtered signal. Next it analyzes the candidates against several criteria and keeps those meeting the criteria. Finally, it finds the positions of the QRS complexes on the ECG signal based on the positions of the accepted candidates.

The Wisconsin algorithm is attractive for heart rate measurements for several reasons. First, it finds peaks rather than thresholds and thus has greater accuracy and wider dynamic range. Second, it detects arrhythmias as well as normal ECG signals. Third, it filters out noise and wander. Fourth, it is a real-time algorithm. Such an algorithm is also needed for non-real-time applications such as processing a file of data too large to fit in system memory at one time.

The two block diagrams in the Hamilton-Tompkins article that illustrate the algorithm's overall signal flow and filter architecture resemble graphical LabVIEW programs. Because of this similarity, I felt that it should be possible to create a LabVIEW version quickly using the article as

reference. It took about three to five days to create the first version along with a data acquisition and display program to test and demonstrate it.

The LabVIEW version of the Wisconsin QRS detector described here is based on the Hamilton-Tompkins design but is not a direct translation of their C program. Therefore, several implementation differences exist. The C program is designed for minimum processing overhead. It therefore uses integer arithmetic and its routines are tailored for speed. The LabVIEW version is designed to be understood and modified easily. It therefore uses as few custom-made routines and as many general-purpose library routines as possible. The C program runs considerably faster, but the speed of the LabVIEW version should be acceptable to many users. Other differences are described at the end of this article.

Because of implementation differences, the sensitivity and predictivity of the LabVIEW version is probably different than the original. How different is not yet known. The LabVIEW version has not been tested against the MIT/BIH database. I expect to make changes as users report their results. Of course, users may make their own modifications—a fundamental feature of LabVIEW.

While the Pau-Tompkin and Hamilton-Tompkins articles describe generally how the Wisconsin algorithm works, this article describes specifically how to implement the algorithm in the LabVIEW programming language, all the way to the lowest-level routine. Notes appear throughout the article to explain basic LabVIEW concepts and terminology that may be unfamiliar to readers. I encourage you to start by perusing the illustrations to see how much you can glean from them alone.

LABVIEW QRS DETECTION PROGRAMS

The remainder of this article describes four LabVIEW programs, or virtual instruments.

> [**Note for readers unfamiliar with LabVIEW:** LabVIEW programs are named **virtual instruments** or **VIs**, because they have the architecture of real instruments and are used like them. Just as traditional programs can *call* subprograms or subroutines, VIs can call other VIs, in which case the term **subVI** is used for the subordinate VI. A **library subVI** is a general-purpose subVI that is either distributed with LabVIEW or available from National Instruments; in other words, one you do not have to create, but that you can modify if necessary.]

The first VI described is **Real-time QRS Detector**, which is a data acquisition, analysis, and display program. This VI measures an ECG signal using either of two National Instruments' plug-in A/D boards for the Macintosh computer: the 12-bit NB-MIO-16 or the 16-bit NB-MIO-16X. The VI calls several subVIs. Among them is the focus of this article, **Wisconsin QRS Detector**, which is described next. Two of its subVIs that are specially created for this application are described last. They are **Wisconsin QRS Filter** and **Keep QRS? /r.**

###

Real-time QRS Detector: An Acquisition, Analysis, and Display Program

Figure 1 shows the front panel of the top-level VI **Real-time QRS Detector**. Figure 2 is the VI's block diagram.

> [**Note:** The **front panel**, which is the graphical user interface of a LabVIEW VI, looks and functions like the front panel of an real instrument. The *STOP* and

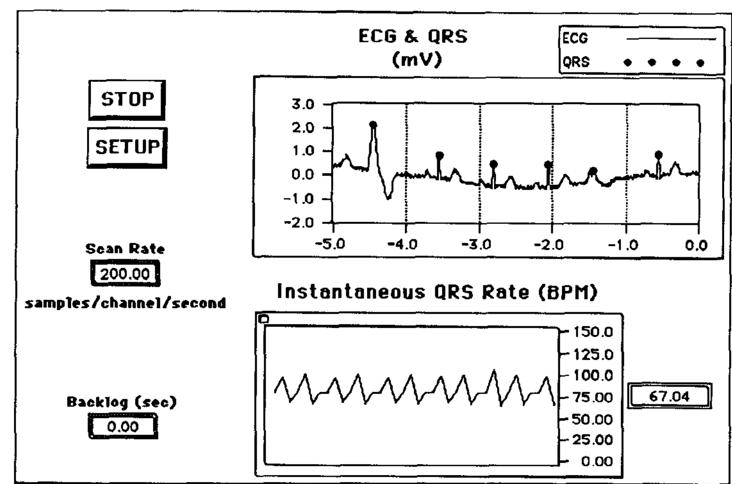


Figure 1. The front panel of the top-level VI Real-time QRS Detector shows a five second ECG history with QRS markers and a longer history of QRS rates.

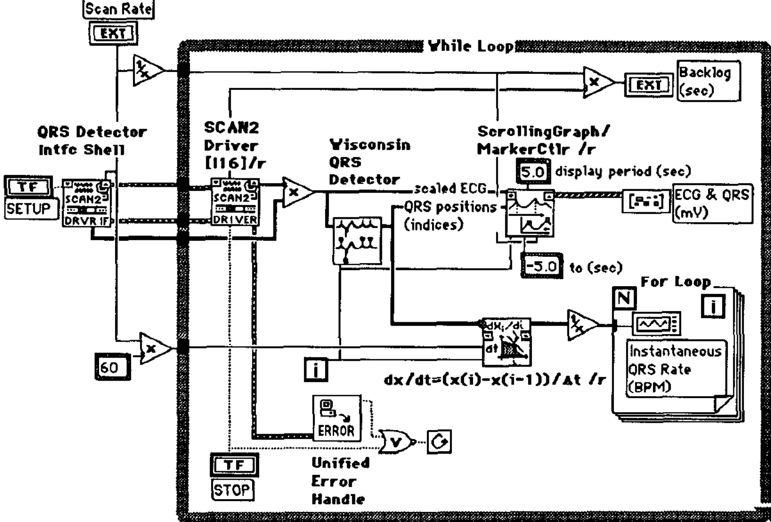


Figure 2. The block diagram of Real-time QRS Detector is the program's graphical source code. This VI measures an ECG signal, scales the measurements, locates QRS complexes, computes the instantaneous heart rate, and plots the results. Towards left-center is the Wisconsin QRS Detector subVI, the focus of this article.

SETUP controls in Figure 1 behave like momentary push-button switches. The *Scan Rate* and *Backlog* indicators imitate digital panel meters. The graph labeled *ECG & QRS (mV)* acts like a scrolling block-mode strip chart. It displays the most recent five seconds of EGG signal with markers on the QRS complexes. The strip chart below it shows a longer history of the computed instantaneous heart rate. It also has a numeric display showing the most recent rate.

The **block diagram**, which is the VI's source code, looks like the schematic diagram from which real instruments are manufactured.

The **terminals** in Figure 2 such as SETUP TF and ECG & QRS (mV) represent front panel controls and indicators. These are ports for variable data to enter and exit the program. **Constants** enter from terminals such as -5.0. The lines between objects, called **wires**, are the paths for data as it flows between the terminals and other objects. Solid wires represent numbers; the dotted wires represent Booleans; and the third style is a cluster or structure of different data types. Thin wires are scalar quantities, and thick one are one-dimensional (1D) arrays. The six square icons such as the one for Wisconsin QRS Detector represent calls to subVIs. The smaller icons that look like arithmetic operators, such as Reciprocal, are built-in **functions**; these become in-line code when the program is compiled. The program uses two program control **structures**, represented by frame-shaped objects encompassing other objects. The larger one is a While Loop, and the smaller one is a For Loop.

The program executes on the principle of **dataflow**: Each subVI, function, or structure executes when all the data it needs arrives on its input wires, and each produces data on its output wires when it completes execution. SubVIs and structures not connected by wires can execute in parallel.]

The first data source is the SETUP terminal at the left side of Figure 2. Its Boolean value passes immediately to the **QRS Detector Intfc Shell** subVI SCAN2 DRVR II. If SETUP is TRUE, a configuration subVI (Figure 3) opens.

With this front panel the user identifies which A/D board and channel the top-level VI will use to measure the ECG signal, the channel gain and external sensitivity, and how the board is configured. The user also selects the update rate, which specifies how often data is acquired and processed. An update rate of 1 means 200 samples are processed as a unit each second. The subVI checks that the selections are valid, informs the user of the status, and computes several parameters needed in the top-level VI. When the user clicks on CONTINUE or CANCEL, the subVI closes and program control returns to the top-level VI.

The configuration subVI returns four data elements to the top-level VI. One of them is the scan rate, which is fixed at 200 scans/second. (A scan is one sample/channel.) After the scan rate is inverted and then multiplied by 60 (seconds/minute) to obtain a scaling factor, the While Loop can begin. The objects in the While Loop structure execute once per iteration until the user clicks the STOP button or an error occurs.

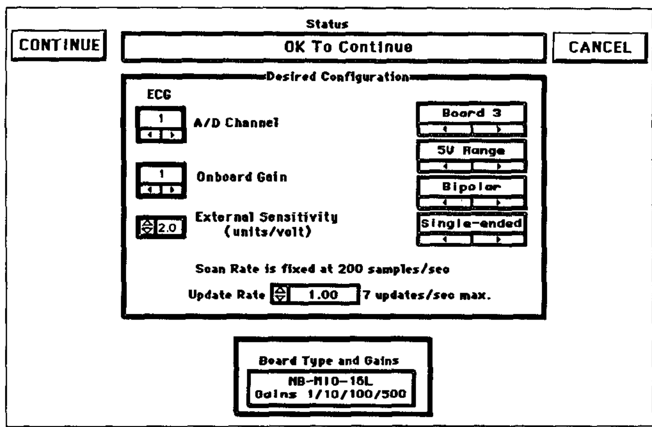


Figure 3. When Real-time QRS Detector begins, this configuration subVI front panel appears on request for the user to adjust configuration parameters.

Inside the While loop, the library subVI **SCAN2 Driver [I16]/r** (or simply SCAN2 driver) executes first. Using parameters computed in the configuration subVI, the SCAN2 driver initializes and starts a continuous acquisition process to measure the ECG signal and place the results in a circular memory buffer. On each iteration, the driver retrieves a segment of binary data from the buffer, then scales the segment to source-level units (mV) by multiplying it with a scaling array created by the configuration subVI. The scaled segment, named ECG[], then flows to the

Wisconsin QRS Detector subVI (the QRS detector).

The QRS detector finds the position of each QRS complex in ECG[] and places an element in the output array QRS Indices[] for each complex found. Depending on how many ECG[] samples are processed in one loop iteration, the output array may be empty. Also, the output may identify a complex that actually occurred in a previous segment but could not be confirmed until this one. For example, a peak may occur in segment 8, but the event qualifying it as a peak might not occur until segment 9 or 10.

The library subVI **dX/dt=(X(i)-X(i-1))/Δt /r** differentiates the indices array using a Δt of 200 samples/second multiplied by 60 seconds/minute to produce an array whose values are the number of minutes between successive QRS complexes or beats. Taking the reciprocal of the array yields beats per minute (BPM).

This BPM array flows to a For Loop containing a strip chart terminal. The loop iterates as many times as there are elements in the array. Notice how the thick array wire changes to a thin scalar wire at the For Loop border in Figure 2. This change indicates that the loop automatically indexes or dissembles the array before plotting the elements point-by-point on a strip chart.

Meanwhile, the arrays ECG[] and QRS Indices[] flow to **ScrollingGraph/MarkerCtlr /r**. This library subVI manages its associated graph like a block-mode strip chart. Internally, the subVI remembers the most recent five seconds' worth of ECG and QRS data. When new data arrives, the subVI discards old data and plots the ECG signal with QRS markers.

As mentioned earlier, the While Loop iterates until the user stops the program or the SCAN2 driver detects a data acquisition error, in which case, the driver automatically stops the acquisition and sets an error flag and related parameters. The library subVI **Unified Error Handler** describes any error that occurs in a message to the user.

> [**Note**: The notation /r in the names of several of the library subVIs used in this application means that each subVI has local static memory and is configured to be reentrant. Each call to a reentrant subVI has its own data space. Thus, a VI can make multiple parallel calls to a reentrant subVI without danger of one call corrupting the data of another. The two arrow glyphs you see in the icon of a subVI are visual reminders that it has static memory. Notice the wire from the While Loop iteration count terminal [i] to several of the subVIs. On the first iteration of the program, when [i]=0, each subVI with static memory initializes that memory before proceeding.]

Let's next study the QRS detector subVI more closely.

Wisconsin QRS Detector

Figure 4 is the block diagram of the Wisconsin QRS Detector subVI. As a reminder, the top-level VI calls this subVI once each iteration to locate QRS complexes in segments of ECG[] data.

The detection process has five stages, which are illustrated in the following descriptions:

1. Filter ECG[] to get X[], Y[], and Z[]
2. Find potential peaks on Z[]
3. Find corresponding peaks on X[]
4. Analyze Z peaks[] and keep the best
5. Find corresponding peaks on Y[] and then on ECG[]

Stage 1. Filter ECG[]

Wisconsin QRS Filter

The **Wisconsin QRS Filter** subVI bandpass filters the input signal ECG[] to obtain Y[]. It then differentiates, squares, and smooths Y[] to obtain Z[]. It also differentiates ECG[] to obtain X[]. This operation accentuates QRS complexes relative to T-waves in order to discriminate between the two later. (Note: X,Y, and Z are signal names used in the Hamilton-Tompkins article.) A description of the filter follows this description of the QRS detector.

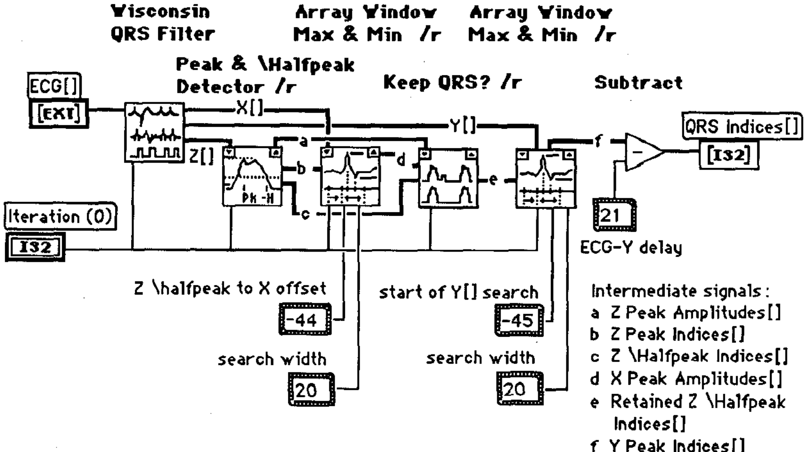


Figure 4. The block diagram of Wisconsin QRS Detector, which Real-Time QRS Detector calls to find QRS complexes.

The cumulative effects of the filter on two ECG signals are shown in Figures 5 and 6. Figure 5 shows the effects from ECG[] to Y[] and Z[] and Figure 6 shows the effects from ECG[] to X[]. The left column of each figure shows a normal sinus pattern, and the right column shows a multifocal pattern riding a wandering baseline. The source for both signals is a Dynatech Nevada 215A Patient Simulator. The unit's left leg and right arm signals are amplified pseudo-differentially by an Analog Devices 5B40-01 wideband, mV input module. The single-ended output signal is measured by an NB-MIO-16 multifunction data acquisition board. **(Note: Neither the amplifier nor the data acquisition board is a biomedical device and neither is approved for direct connection to humans.)**

Stage 2. Find Potential Peaks in Z[]

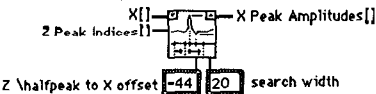
Peak & \Halfpeak Detector /r

Z[] — Z Peak Amplitudes[], Z Peak Indices[], Z \Halfpeak Indices[]

The library subVI **Peak & \Halfpeak Detector** /r (or peak detector) finds peaks on Z[] by designating each local maximum as a peak when the signal level subsequently falls to half the amplitude of the maximum, that is, the half-way point on the falling slope (\halfpeak). The results of this operation are two position arrays named Z Peak Indices[] and Z \Halfpeak Indices[] and an amplitude array named Z Peak Amplitudes[].

Stage 3. Find Corresponding Peaks in X[]

Array Window Max & Min /r



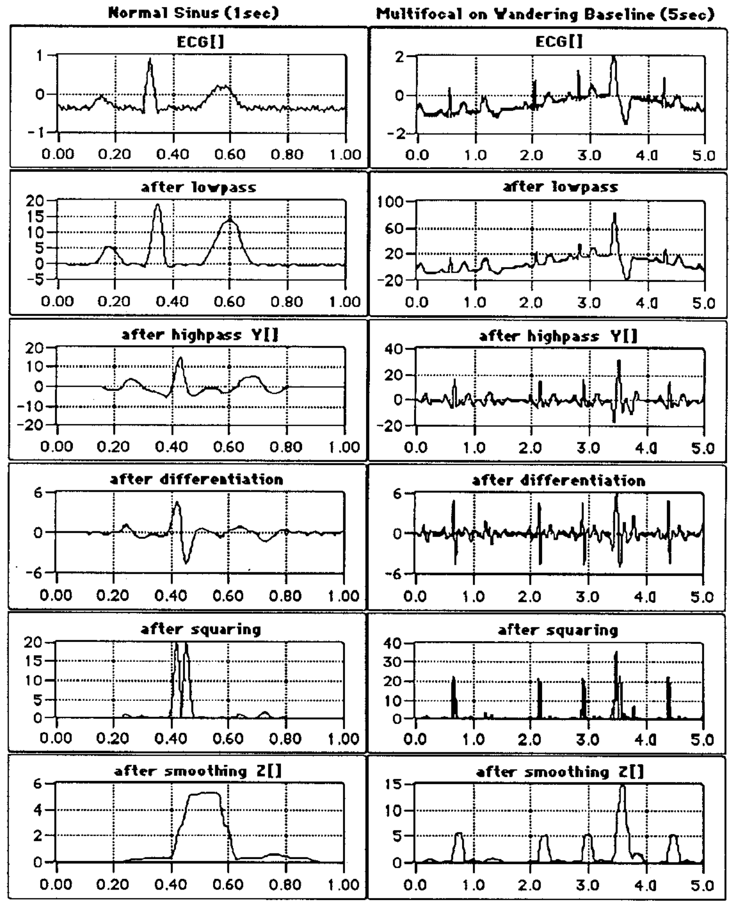


Figure 5. Effects of bandpass filtering, differentiating, squaring, and smoothing the ECG signal.

The library subVI **Array Window Max & Min /r** finds peaks in X[] corresponding to those in Z[]. X[] is a derivative of ECG[] in which QRS complexes stand out relative to T-waves. The filter delay between X[] and Z[] is approximately 34 samples or 170 msec, but noise may cause the maximum to occur slightly off this point. Therefore, this subVI finds the peak by searching for the maximum value within a window 20 samples (100 msec) wide starting 44 samples (220 msec) before the Z halfpeak position, as shown in Figure 7. The result is a position array named X Peak Indices[].

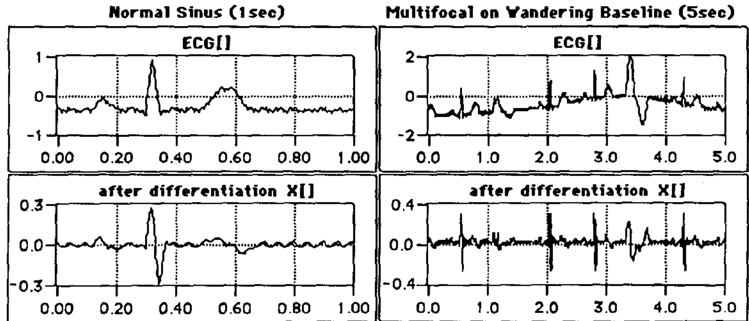


Figure 6. Effects of differentiating the ECG signal.

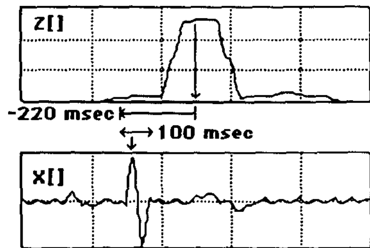


Figure 7. Array Window Max & Min /r finds the location of a peak in X[] knowing the location of the corresponding peak in Z[] and the approximate distance between them.

Stage 4. Analyze Z[] Peaks and Keep the Best

Keep QRS? /r

Z Peak Amplitudes[]
Z \Halfpeak Indices[]
Retained Z \Halfpeak Indices[]

The purpose of the subVI **Keep QRS?** /r (or Keep) is illustrated in Figure 8, which shows six QRS complexes before and after filtering.

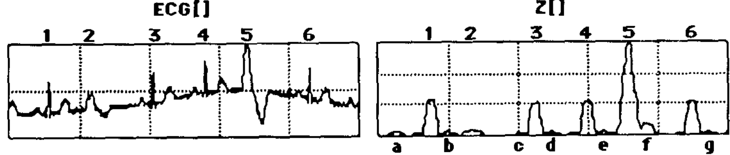


Figure 8. Six true QRS complexes (1-6) before and after filtering, along with seven false potential peaks (a-g) that must be eliminated.

The preceding peak detector subVI finds the six true peaks (1-6) as well as several false peaks (a-g). Keep must not only eliminate peaks that are obviously too low to qualify (a-e and g) but also those (like f) that are higher than true peaks (like 2). It does that by applying the following rules and tests to each potential or candidate peak in Z[]:

a. Accept Primary Candidates. Keep accepts each candidate peak if:

1. Primary Acceptance Test. The candidate's amplitude is greater than or equal to:

NPL + TC*(QRSPL-NPL), where

NPL = Noise Peak Level, the running median amplitude of the last eight rejected Z[] peaks;
TC = threshold coefficient, 0.189;
QRSPL = QRS Peak Level, the running median amplitude of the last eight accepted Z[] peaks.

2. Refractory Period Test. The candidate does not occur within 200 msec of an accepted peak.

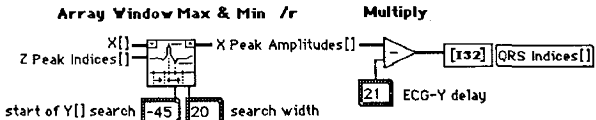
3. T-wave Test. The X[] amplitude of the candidate is equal to or greater than twice the amplitude of the previously accepted peak and the candidate occurs between 200 and 360 msec after the accepted peak.

b. Save Secondary Candidates. The subVI saves all candidates that fail test a1 but pass tests a2 and a3 and are greater than or equal to 0.5 DT.

c. Search Back. If Keep does not find a primary candidate within 130% of the running median of the last eight RR intervals, it accepts the largest of any secondary candidates saved in step b.

The subVI updates the running QRSPL and RR medians upon accepting a candidate and the running NPL median upon rejecting one.

Stage 5. Find Peaks on Y[] and ECG[]



A second call to the Array Window Max & Min /r subVI finds peaks in Y[] corresponding to the accepted Z[] peaks using the window search method employed in stage 3. This time the subVI's Peak Indices[] output is used. In this case, the peak is the larger of the two extrema in the window, the maximum or the absolute value of the minimum. The result is a position array named Y Peak Indices[]. The filter delay between the original ECG[] signal and Y[] is 21 samples (105 msec); therefore, this amount is subtracted from Y Peak Indices[] to get QRS Indices[].

Of the subVIs comprising Wisconsin QRS Detector, only the Wisconsin QRS Filter and Keep QRS? /r are created specifically for this application. They are described next.

Wisconsin QRS Filter

Figure 9 is the block diagram of the multistage **Wisconsin QRS Filter** (the filter). It calls to three library subVIs and a Multiply function.

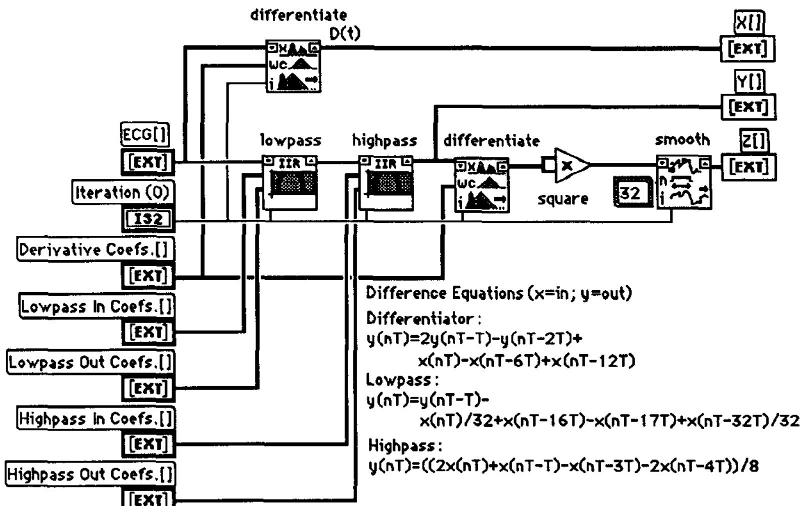


Figure 9. The block diagram of Wisconsin QRS Filter

The main filter branch has five cascaded stages: lowpass filter, highpass filter, differentiator, squaring operation, and smoother or moving averager. The QRS detector that calls this subVI uses the intermediate bandpass signal Y[] and the final signal Z[] to find the QRS complexes. The second branch consists of a differentiator with the same response as the one in the main branch, and the detector uses its output signal X[] to discriminate between T-waves and QRS complexes.

IIR Filter /r performs the lowpass and highpass filtering, and the FIR filter **WeightedMvgAvgr /r** performs the differentiation. The filter coefficients for these stages are derived from the difference equations in the block diagram. **MovingAverager /r** is an FIR filter with equally-weighted window coefficients that performs a 32-point smoothing.

Figure 10 shows the filter's impulse response. The left column shows the response of the individual stages, and the right column shows the cumulative response leading to Y[] and Z[]. The filter is designed for a sample rate of 200 Hz, or a Nyquist frequency of 100 Hz.

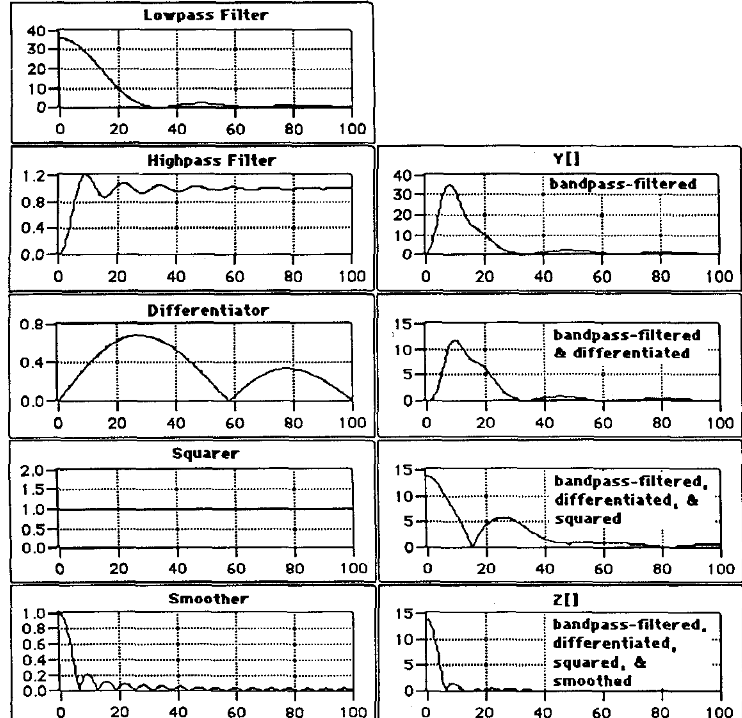


Figure 10. Impulse responses of the Wisconsin QRS Filter.

Keep QRS? /r

The last VI unique to this application is the subVI **Keep QRS? /r** (or Keep), whose block diagram is shown Figures 11 and 12. Keep evaluates each candidate peak and retains those that meet the criteria described earlier.

> [**Note**: The architecture of Keep is a While Loop encompassing a For Loop and several Case structures, which are labeled A-D at their bottom edges. The While Loop holds Keep's static memory elements; the For Loop processes the incoming data, and the Case structures, as the name implies, are used for conditional execution of program code.
>
> Each pair of arrow-shaped terminals on the For Loop and While Loop boundaries in Figure 11 constitute a **shift register**. It is the structure's

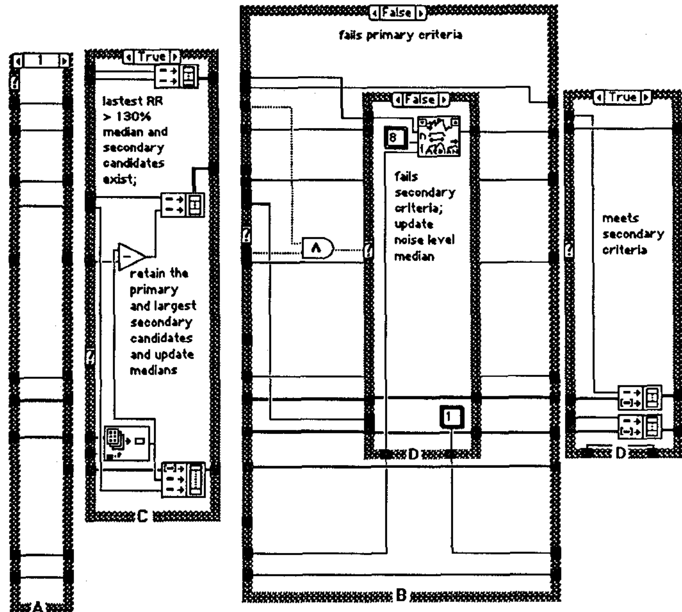


Figure 12. The other Case subdiagrams of Keep QRS? /r.

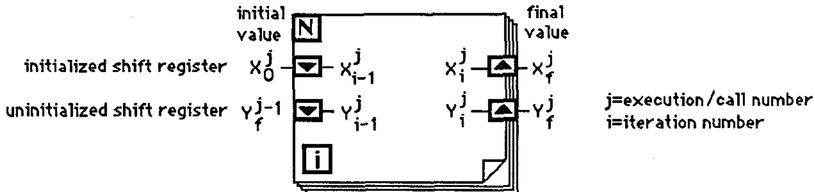


Figure 13. Movement of data through initialized and uninitialized shift registers. The subscript (i) is the iteration number, and the superscript (j) is the execution number.

The first time Keep is called, case 0 of Case structure A executes to supply initialization values or initial conditions for the For Loop. After the For Loop finishes, its output values or final conditions are stored in the While Loop's shift registers. On the next call to Keep, case 1 of A executes and passes those final conditions to the For Loop as its new initial conditions. Case 1 also executes on subsequent calls.]

Figure 14 is an extract of Figure 13a that shows the logic path for the primary acceptance tests 1a and 1c. A candidate peak in Z Peak Amplitudes[] is accepted as a QRS peak if it is greater than equal to NPL+0.189(QRSPL-NPL) and it occurs 73 or more samples (≥365 msec) from the last peak. Constants supply the detection threshold coefficient (0.189) and the QRS to end-of-T-wave distance (73 samples). These and other nearby constants—the searchback multiplier, (0.5), RR delay factor (1.30), end of refractory period (40 samples), and X peak threshold coefficient (0.7), can be altered in order to fine-tune the Keep algorithm.

A library subVI of Keep, **MovingMedianFilter /r**, computes three running medians, one each for the amplitudes of the accepted (QRSPL) and rejected (NPL) peaks, and one for the RR interval of accepted peaks.

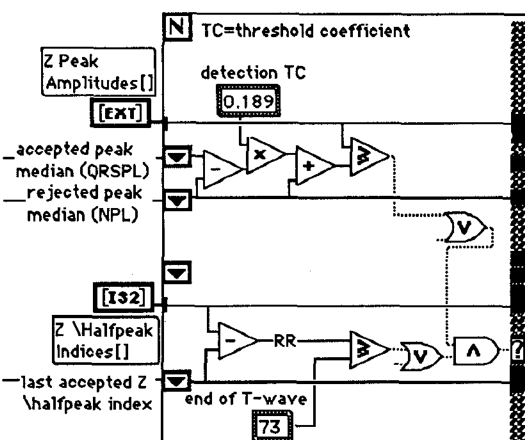


Figure 14. The logic path for the primary acceptance criteria of Keep QRS? /r.

IMPLEMENTATION DIFFERENCES

The Pan-Tompkins and Hamilton-Tompkins versions of the Wisconsin QRS detector are designed for maximum speed. Each takes less than the sample interval of 5 msec to process each point and therefore can process real-time signals sample-by-sample. The LabVIEW version, on the other hand, runs at near-real-time rather than real-time speeds. While it is designed to process segments of any size, the overhead for a segment size of one exceeds 5 msec. The per-sample overhead decreases as the segment size increases. How large the segment must be to keep up depends on the computer. The top-level VI, Real-time QRS Detector, which combines measurement, analysis, and display, can run at seven segments or updates per second on a Macintosh IIfx computer. That computes to a segment size of about 30 samples and a latency between event and display of between 150 and 300 msec. On the slower Macintosh II, the fastest update rate is about one segment per second, leading to a latency of between one and two seconds.

The LabVIEW version of the Wisconsin QRS detector is not constrained to use integer arithmetic as are the two original versions. Therefore, several threshold values differ. For example, the C program uses a detection threshold coefficient of 0.1825 ($2^{-3}+2^{-4}$), whereas the LabVIEW version uses the desired threshold of 0.189. The maximum delay between RR intervals before a secondary candidate is chosen is 130% of the median rather than 150%.

The peak detector used in the C program to find potential peaks after filtering accepts a local maximum as a peak after 170 msec (35 samples) if the \halfpeak point is not reached by that time. This procedure is meant to detect wide QRS complexes. The LabVIEW peak detector does not do this because it is not appropriate for a general-purpose library subVI. Another way to process wide complexes is to adjust the offset and width parameters at Stage 3 of the detection process depending on the distance between peak and \halfpeak. These parameters are controlled with constants in Figure 4.

When locating the QRS peak on Y[] after keeping the best candidates from Z[], the C version accepts a negative peak if its magnitude is greater than twice that of the positive peak in the window. The LabVIEW program weighs each polarity equally.

CONCLUSION

QRS detection of the type described in this article is a complex operation involving many processing stages. While LabVIEW is not the only way to implement such an algorithm, it does have several advantages over other approaches. First, LabVIEW is a fast way to program, especially for engineers and scientist who are not programmers or only part-time programmers. If you typically design an application in diagram form, then convert the diagram to a text-based program, LabVIEW lets you combine the two steps into one. Furthermore, creating sophisticated and easy-to-use graphical user interfaces to LabVIEW programs requires no programming whatsoever. Second, a LabVIEW program is easier to understand than one written in a conventional language because the source code is a diagram. For example, the block diagram of the Wisconsin filter shown in Figure 9 is nearly identical to the filter diagram in the Hamilton-Tompkins article. The LabVIEW diagram is both an informative picture of the process and the program that implements the process. Third, the LabVIEW program is easier to modify by someone who did not create it than a convention program is. Again, the pictorial nature of the program makes this possible. Finally, LabVIEW is supported by a wide assortment of data acquisition, instrument control, and digital signal processing hardware and software—all the pieces needed to put applications like this together.

REFERENCES

[1] Pan, J. and Tompkins, Willis J., A real-time QRS Detection algorithm, *IEEE Trans. Biomed. Eng.*, **BME-32**(3):230-236, 1985.

[2] Hamilton, Patrick S. and Tompkins, Willis J., Quantitative Investigation of QRS Detection Rules Using the MIT/BIH arrhythmia Database, *IEEE Trans. Biomed. Eng.*, **BME-33**(12):1157-1165, 1986.