1-1-2010

The Design and Implementation of an Extensible Brain-Computer Interface

Scott Burns
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS
School of Engineering and Applied Science
Department of Biomedical Engineering

Thesis Examination Committee:
Dennis Barbour, M.D. Ph.D. Chair
Dan Moran, Ph.D.
Baranidharan Raman, Ph.D.

The Design and Implementation of an Extensible Brain-Computer Interface

by

Scott Burns

A thesis presented to the School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

May 2010
Saint Louis, Missouri

ABSTRACT OF THE THESIS

The Design and Implementation of an Extensible Brain-Computer Interface

by

Scott Burns

Master of Science in Biomedical Engineering

Washington University in St. Louis, 2010

Research Advisor: Dennis Barbour

An implantable brain computer interface (BCI) includes tissue interface hardware, signal conditioning circuitry, analog-to-digital conversion (ADC) circuitry and some sort of computing hardware to discriminate desired waveforms from noise. Within an experimental paradigm the tissue interface and ADC hardware will rarely change. Recent literature suggests it is often the specific implementation of waveform discrimination that can limit the usefulness and lifespan of a particular BCI design. If the discrimination techniques are implemented in on-board software, experimenters gain a level of flexibility not currently available in published designs. To this end, I have developed a firmware library to acquire data sampled from an ADC, discriminate the signal for desired waveforms employing a user-defined function, and perform arbitrary tasks. I then used this design to develop an embedded BCI built upon the popular Texas Instruments MSP430 microcontroller platform. This system can operate on multiple channels simultaneously and is not fundamentally limited in the number of channels that can be processed. The resulting system represents a viable platform that can ease the design, development and use of BCI devices for a variety of applications.

Acknowledgments

First and foremost, I'd like to thank my thesis advisor, Dr. Dennis Barbour, for the opportunity to spend the previous two years performing research in his laboratory. Along the way, I have accrued much knowledge about science and engineering, none of which would be possible without his assistance.

I would next like to thank my fellow co-workers. Paul Watkins and Woosung Kim provided all the guidance and help one can expect from older students. I would also like to thank Kim Kocher for her help in all non-research matters.

Finally, I would like to thank my family. To my parents, without whose support I would not be the man I am today. I am forever grateful for your continual guidance and love. To my future wife, I thank you for believing in me when I did not. Please accept my most sincere appreciation.

Scott Burns
Washington University in St. Louis
May 2010

Table of Contents

List of Figures

Chapter 1 Introduction

Connecting the mind to computer understandably evokes thoughts of the future. The technology has been a mainstay in modern science fiction and popular motion pictures. Beyond popular culture though, the ability to process signals from the brain quickly and accurately represents a platform for scientific research and clinical therapy without peer. Towards this goal, the output of brain-computer interface (BCIs) research has grown enormously over the past decade. In addition to restoring hearing (Wilson, Finley et al. 1991), relieving Parkinson's disease symptoms (Obeso, Olanow et al. 2001) and treating depression (Bewernick, Hurlemann et al. 2009), newer clinical applications of BCIs include restoring speech capabilities (Kennedy, Bakay et al. 2000) and providing motor control of prosthetics (Hochberg, Serruya et al. 2006). Scientifically, both invasive and non-invasive BCIs allow researchers to study the brain in a more natural state by removing subject immobilization requirements traditionally necessary when recording the electric potential from individual nerve cells. The promise of these devices, both clinically and scientifically, grows both with the development of new electronics technology as well as knowledge of how the brain functions normally and can be manipulated electrically.

2.1 Origins

The first traceable mention of brain-computer interfaces in the scientific literature dealt with using EEG traces to derive control signals from the brain(Vidal 1973). EEG continues to provide useful (albeit low-dimensional) control signals, but the paper also accurately depicts how the field would develop in the future. Specifically, Vidal foretold of both scientific and clinical applications through brain-computer interfaces. Unfortunately for Vidal, the field lacked the technology necessary to tackle the significant challenges present in the early 1970s.

As technologies improved, several groups used brain-computer interfaces to effectively "read" motor regions of the mind (Schmidt, McIntosh et al. 1978; Georgopoulos, Lurito et al. 1989). Another group derived a 1-dimensional control signal from EEG traces to move a computer cursor (Wolpaw, McFarland et al. 1991). Many groups since have used both invasive (Kennedy, Bakay et al. 2000; Taylor, Helms Tillery et al. 2002; Santhanam, Ryu et al. 2006) and

non-invasive technologies (Utsugi, Obata et al. 2007; Huang, Lin et al. 2009) to derive control signals used to operate prosthetics and computer interfaces. Much of the early progress of brain-computer interface technology can be attributed to the laboratories interested in the motor cortex region. Indeed, an early and continued interest in brain-computer interfaces is the restoration of function by routing control signals around immobilized limbs (Kennedy, Bakay et al. 2000; Taylor, Helms Tillery et al. 2002).

The majority of published modern implantable BCI designs involve the recording of neural signals and their transmission to a remote storage device (often a personal computer) for offline analysis. Early designs implemented a relatively small number of recording channels (Grohrock, Hausler et al. 1997; Lei, Sun et al. 2004), while recently published devices provide data acquisition with relatively high channel density (Yun, Kim et al. 2007; Chestek, Gilja et al. 2009; Rizk, Bossetti et al. 2009). Other designs have provided a means to remotely stimulate neurons (Xu, Talwar et al. 2004). While providing fruitful research, this type of BCI design is limited by one-way communication, particularly poor for closed-loop applications.

2.2 Current State-of-the-Art

Until the 1st decade of the 21st century, brain-computer interface designs primarily focused on solely capturing information from the central nervous system. Mavoori, *et al.*, presented the first implantable BCI incorporating closed-loop, autonomous action potential detection and neural stimulation: the Neurochip (Mavoori, Jackson et al. 2005). The Neurochip's novel engineering provided the enabling technology for an important discovery regarding cortical plasticity (Jackson, Mavoori et al. 2006). The low-power 8-bit microcontroller used in the Neurochip was relatively constrained in processing capability, restricting closed-loop applications to neural recording on a single channel and ultimately limiting the general utility of the design. Hampson, *et al,* have recently presented a recording device built with a microcontroller capable of recording and wirelessly transmitting 16 channels of neural data across a Bluetooth interface (Hampson, Collins et al. 2009). This device does not currently provide control signals to other pieces of hardware, limiting the designs ability to transmit information into the central nervous system.

Distinct from the use of off-the-shelf electronics hardware, a productive avenue of research has concentrated on the design of application-specific integrated circuits (ASICs) specifically designed to detect action potentials or other signals of interest and wirelessly transmit information (Mojarradi, Binkely et al. 2003; Harrison, Kier et al. 2009; Raghunathan, Gupta et al. 2009). Ultra low power requirements and small size are major advantages of BCI designs employing ASICs; however, once designed and implemented in silicon, these devices cannot be customized with added functionality without redesigning the hardware circuits. If the brain-computer interface application requires only capturing the neural waveforms and not generating any stimulus in response, these ASIC designs represent the state-of-the-art in recording capabilities. Such open-loop design may limit both scientific and clinical application, however.

2.3 Need for Novelty

There currently exists no general solution for extracting multichannel closed-loop control signals from neural data in low-power, general-purpose BCIs. The preferred design for such a system would easily scale to handle the number of channels required by the scientific experiment or clinical application. The overall design should also adapt easily to different experimental paradigms. In this thesis, I present a BCI design that balances flexibility and energy-efficiency yet can execute multiple channels of closed-loop signaling and control. This work has been performed in parallel with a scientific project investigating the ability to reorganize small cortical networks based on artificially-induced spike-timing dependent plasticity.

A brief report of these results has appeared previously in abstract form (Burns and Barbour 2009).

Chapter 2 Design and Implementation

The work presented in this thesis covers two bases-firmware and hardware. The hardware consists of a headstage, signal conditioning circuitry, and computing hardware to discern neural waveforms of interest from noise. The firmware is an abstract design facilitating waveform discrimination, program flow, and data storage implemented in the C language. Together, these two pieces form the basis for an extensible low-power BCI.

2.1 Firmware

In the design of digital computing systems, there is a distinct separation of role and facility between hardware and software. Hardware provides the actual computing logic with which algorithms are processed. In a modern desktop personal computer, hardware components include the central processing unit (CPU), external memory (either read-only memory (ROM) or random-access memory (RAM)), input/output peripherals such as serial transceivers and graphics displays and power management devices, just to name a few. Typically, the desktop PC runs an operating system (OS) to intelligently manage the hardware and provide an abstraction of the hardware, easing the development of user applications. The OS and user applications can be changed at will and are thus considered software.

It is more difficult to define firmware. Like its name suggests, firmware exists between the hardware and software, but even this statement depends on the application. In desktop PCs, firmware can be found in the booting hardware (Doran 2003) and contains code to initialize important hardware components and ensure proper loading of the OS, which then assumes control the hardware until shutdown. After the PC has been built and shipped, the user will rarely need to change this firmware, though many operating systems do provide the ability. In this instance, firmware acts more like hardware.

In embedded applications though, firmware takes on a role more similar to software. The exact definition of an embedded application is difficult to pin down, but these applications can be thought of products requiring computing resources while the user not recognizing the device as a typical computer (Edwards, Luciano et al. 1997). Examples of typical embedded applications are automated teller machines, home DVD or stereo systems, remote controls and the engine controller module found in all modern automobiles. These systems contain firmware implementing required functionality in the same way that individual applications do on a PC. The firmware runs in a self-contained environment and cannot easily be changed in production systems without reprogramming. Programming firmware into a computing environment requires a host computer to physically transfer the binary code into memory and can be extremely costly or impossible once a product has shipped, depending on the hardware.

Wherever firmware lies along the hardware-software continuum, it must absolutely be robust, error-tolerant, and if possible, self-healing. It is notoriously difficult to diagnose and fix errors yet integral to any application requiring digital computing resources.

Like software and hardware, there are many engineering decisions when designing and implementing firmware. Factors dependent on these choices include speed of the compiled code, responsiveness to external signals, program size (when actually implemented in the processor), source code maintainability and program extensibility. With constrained resources, the engineer cannot maximize all of these qualities at once but must balance these with the demands of the application.

Perhaps the first and most important choice when designing firmware is architecture. Laying a strong foundation allows the firmware to accumulate functionality over time with minimal restructuring. Luckily, there are few options for this choice. Firmware is generally architected in one of two ways – on top of a real-time operating system (RTOS) or as a run-loop (RL).

A RTOS is designed with the same goals as a desktop OS. Generally, the RTOS abstracts the

hardware by providing resources for computing to the programmer. RTOSs often define data structures such as threads, queues, semaphores and timing capabilities much like desktop operating systems. On the other hand, because they are targeted at much smaller processors (both in physical size and memory), real-time operating systems must be orders of magnitude smaller in code size than their desktop brethren. As such, real-time operating systems lack much of the functionality taken for granted in modern desktop operating systems such as a choice of file systems, graphical capabilities or wide peripheral support simply because these requirements are not needed in a small embedded application. Real-time operating systems are often very efficient and extremely responsive to external events. This responsiveness to events garners the "real-time" prefix. These systems can be designed to deterministically operate within certain time constraints, ensuring event processing will occur within a known period after the event occurs. Real-time operating systems provide the same layer of abstraction as the desktop counterpart, thus ensuring code written against particular real-time operating systems will run on any hardware platform the real-time operating systems supports.

There are two major disadvantages to using real-time operating systems, however. Layers of abstraction, while beneficial to the programmer, create figurative levels between the "iron" (e.g. actual hardware) and code and can decrease performance. This disadvantage is particularly troublesome when designing applications such as the current one where fast code execution is vital. The second disadvantage is that real-time operating systems increase the size of the firmware. Modern compilers are adept at minimizing data structures and the size of outputted code, but it should be obvious that having to compile an RTOS along with the application code will increase the total programs size. This is becoming less of an issue as memory capacity increases in microprocessors and microcontrollers, however.

A run-loop based architecture is much simpler than that of a real-time operating system. Many simple embedded applications can be modeled as simply a system waiting for some event (a timer, user input, etc.), executing code when the event occurs, and then sleeping until the next event. From this description, the run-loop metaphor should be obvious. The program idles at a place in code until an event occurs, the event is handled, and the code

returns to an idle state.

The run-loop architecture requires very little overhead, so the application code has almost full use of all hardware resources. Because relatively few events are being monitored, firmware with this architecture is as responsive to external events as the processor itself. The little code overhead to maintain the loop results in very small code bases and also helps in source code maintainability. Because the programmer knows explicitly which code will execute at each event, the program can be designed orthogonally.

The run-loop architecture is ideal for small, focused designs in which few events need monitoring. As the variety of events grows in an application, the problem of scheduling (making intelligent decisions about what code to execute when) increases, especially when events occur concurrently. Many combinations of events cannot be adequately tested, leading to obscure firmware bugs appearing only in production code. Real-time operating systems excel when the application requires monitoring of many events while run-loops are more ideal for simpler applications. Because of the scheduling problem, run-loop architectures can be extended only so far before a real-time operating system should be considered.

Compared to other digital electronics, a processor for discriminating neural signals is relatively simple. Obviously, neural signals recorded from the brain must be transformed (at some specific sampling rate) into digital signals for manipulation in microprocessors. The sampling event is a natural place to awaken the device and discern the signal for a waveform of interest. It is also the only event that needs to be monitored. Therefore, the firmware in this thesis is based on the run-loop architecture.

To increase usability among different research groups, the firmware has been factored into two parts. The first part is a library providing a framework for organizing incoming neural data and how to process such data. The second portion is an application built upon this library for detecting single-unit action potentials with no user interaction.

2.1.1 Library

Requirements

As mentioned earlier, any abstraction can decrease code performance. It is the price programmers pay when using high level languages. C code is slower than pure assembly language and interpreted languages such as Perl or Python are slower than C code. When portions of a program require the utmost performance characteristics, lower-level languages should be used.

Often-used functions or capability are often packaged into groups of source code termed libraries. Libraries can implement many different kinds of functionality in any language, and newly written programs can a library's functionality without "having to re-invent the wheel". Libraries expose their capability through a set of classes or functions named the API (Application Programming Interface). The benefit to the library writer is that he/she need only maintain the API from revision to revision and users can upgrade to better versions without having to change existing code. The benefit to library users is that they need not worry about the internals of library, merely that calling the functions will have their intended (and hopefully documented) consequences. It should be obvious that libraries are a useful tool for programmers.

A product of this research thesis is a library implementing a minimum set of requirements to be useful to other engineers implementing brain-computer interfaces.
This library provides resources for storing incoming neural data and provides facilities to process the data in an arbitrary manner. The data storage techniques should maintain performance across large and small buffers. The overhead for maintaining data processing functions should be lightweight (i.e. there is little penalty for calling the data processing functions) and should facilitate swapping processing functions as neural signals are non-stationary and different characteristics of the neural waveform may call for different processing techniques. Ultimately, the library should be easy to use and provide a simple, well-designed API.

Buffer

Brain-computer interfaces, no matter the application, deal with a multitude of incoming data. Almost universally, the data takes the form of electrical signals arising from cells in the central or peripheral nervous system. These analog electrical signals are transformed through microelectrodes, amplified, and mapped into the digital domain with specialized Analog-To-Digital circuitry. Once in the digital domain, the neural signals are more easily manipulated and useful in computation.

However, our nervous systems are constantly producing these signals, so they must be stored for sometime, processed, and then either saved for future analysis or discarded. In computing terms, they must be placed into a buffer. A buffer is a data structure designed to store disparate pieces of data, maintain logical order between all the pieces of data and facilitate data processing. In this application, the incoming data is logically ordered by the time at which the signal occurred; thus the buffer stores a finite amount of previously recorded neural data.

Depending on the processing involved on the data (which may vary from application to application), the required size of the buffer will vary. Some time-amplitude algorithms may only require a size approaching the temporal width of an action potential, while more complex algorithms require a larger buffer. No matter the size, the buffer should deterministically provide constant times for insertion and traversal throughout its use in the program.

A buffer is inherently a type of list of which there are many implementation choices. The major delineation between list implementations is static data or dynamic data. Static data is stored on the program stack and therefore access time is very fast. However, stack variables must be declared at compile time and cannot change in size during the course of the program at run time. Dynamic data types are stored in the heap, a block of memory used by the C runtime system. These variables are allocated and de-allocated during run time and can therefore grow or shrink as new storage is needed. However, using variables in the heap requires a pointer dereference, thus incurring a time penalty. Pointer dereferencing is extremely fast on modern hardware, but the penalty still exists. Also, allocation and de-

allocation can be very time-consuming, so these routines should be kept to a minimum.

If code execution speed is important and data structure size is known at compile time, then a static structure should be used, but this has caveats. A high-level language such as C allows the usage of functions, or blocks of code declared once and used throughout the program. Functions receive arguments and may return outputs; most non-trivial functions will receive some data structure as input. The C language uses a function evaluation strategy known as "call-by-value". The caller (the piece of code using a particular function) passes arguments to the function, and the runtime system copies these arguments into a new memory block only the function can use. When the function completes, this memory block is discarded and the return variable is copied to the caller.

This evaluation strategy increases code security by hiding variables only used in a function, but the copying of a large data structure (in this case, the buffer) can also severely degrade code performance. As such, dynamic memory is often used in large systems because functions only need to copy the pointer variable that tracks the data structure in the heap.

Because data will constantly be inserted and removed from the buffer in this application, continually allocating memory for each datum inserted and de-allocating the removed datum point would incur stresses on the runtime system, also decreasing performance. A more optimal design is to allocate all memory at system start up and merely re-write the newer data at indices holding older data. This design is known as a circular buffer. There is no explicit beginning or end of the buffer, but rather pointers are used to track the newest and oldest data points. To insert a new data point, the piece of memory referenced by the oldest pointer is overwritten, and the pointers are incremented. Because the only cost of insertion in a circular buffer is the pointer increments and is not related to the size of the buffer, this runs in O(1) time.

To dereference the buffer in a temporally coherent manner (that is, to "view" the buffer from the oldest datum to the newest), a temporary pointer can dereference the beginning pointer's

data and increment/dereference the next point until the temporary pointer equals the pointer referencing the newest datum. Like any list traversal, this requires O(N) time.

However, computer memory is 1-dimensional. To lookup a piece of memory, the only information required is the address. When an array is allocated, the returned piece of memory is a contiguous block of memory. To add the circular semantics of the buffer described above to the linear array, some logic is required. During the insertion, if either pointer references a piece of memory beyond the range of the array, the pointer should be changed to point to the beginning of the array. This branching logic incurs a slight performance penalty that will actually decrease as the size of the buffer increases. That is, for large buffers, the "wrapping" of the pointer from the end to the beginning of the array will occur less often. The pointers tracking the beginning and end of the buffer as well as the end of the array represent some storage overhead in the buffer, but this overhead is minimal to capture the circular buffer semantics. Some data storage structures include methods for removing data, often to minimize required storage. Because data is continually written over in the circular buffer and the only memory ever needed is allocated at once, there is little need for this functionality. The methods and design of the circular buffer is depicted graphically in Figure 2.1 below.

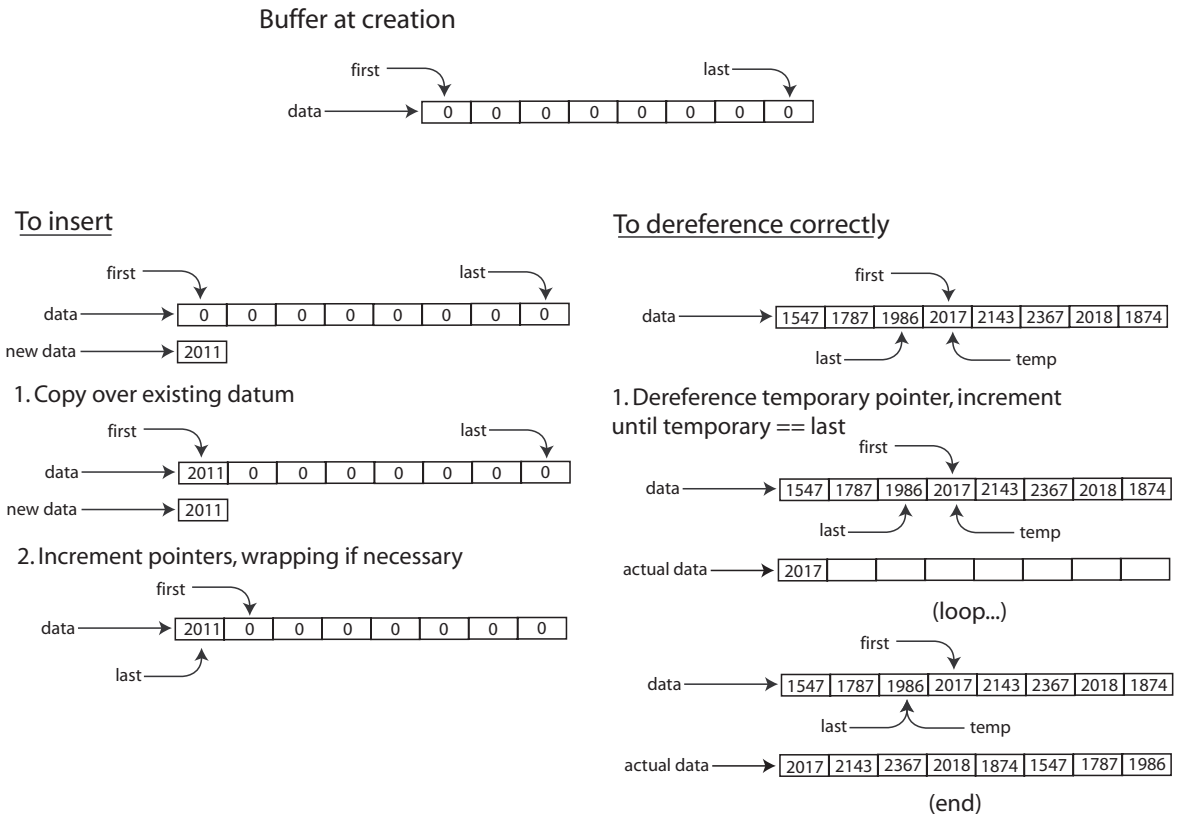

Figure 2.1. Buffer Design and Algorithms

Graphical depiction of the design and methods of the buffer.

Channel Structure

In BCI applications, the channel structure abstracts specific data converted from a specific hardware component (often a microelectrode) and how the channel should be processed. In purely recording applications, the converted data might be the only required information to retain. For example, incoming data could be instantly transmitted to some external data logging device (a personal computer) and no processing is required on the BCI itself. However, there is a trend towards closed-loop applications (Jackson, Moritz et al. 2006)

One can imagine the signal processing occurring on the data logging machine and any detected control event could be relayed back to the BCI in a sort of server/client configuration. In this configuration, there is transmission overhead that places an upper bound on the responsiveness of the system to biological events. For applications that do not require high levels of responsiveness, this may be acceptable. However, for closed-loop BCI applications that require almost real-time responsiveness to biological events, signal processing must be performed on the device itself. Microprocessor performance per area has exponentially increased for many generations of devices and is predicted to increase in such a fashion for at least the next decade (Moore 1965). What was not possible a decade ago regarding onboard signal processing is now available.

Because microelectrodes inserted into the brain will pick up units of varying levels of isolation, it is important to process each channel independently. The waveforms of interest will vary in shape on each microelectrode, so while the same algorithm may be used to detect waveforms across all channels, each instance of a channel should use its own specific variables. To implement this functionality, a group of function pointers was designed to facilitate any user-implemented waveform detection algorithm.

In the same way pointers are used to reference data not explicitly declared at compile time, function pointers are used to reference functions not explicitly known at compile time. To the hardware CPU, the only difference between data and instruction is how the memory is

interpreted; this is handled by the compiler. A function pointer is declared as a normal variable but includes syntax for the output type and argument list. This function pointer can be assigned to any declared function whose input and output signature matches those of the function pointer. The function pointer can then be used to call the function instead of the name given by the programmer, which is discarded by the compiler after all.

Function pointers allow for flexibility in code that can benefit performance when runtime information can be used to make more informed decisions about information processing than the programmer can at compile time. There is no performance penalty for using function pointers because most mature compilers treat the function identifier (e.g. the function name) as a function pointer. Function pointers can be changed like any variable, so a program can potentially use one of many implemented functions to perform a task. This system of binding functionality at runtime instead of compile time is commonly referred to as a callback.

The channel structure presented in this work is designed to be used within a run-loop architecture that operates for the entire period the brain-computer interface is operating. For example, each loop iteration could be coupled to an analog-to-digital conversion on every channel. There is a single method associated with the channel structure, *insert_and_process()*. As its name implies, this method receives the channel structure pointer and datum point most recently converted by the A/D circuitry. The method inserts the datum point into the buffer (as discussed in the previous section) and processes the channel with three function pointers.

The first function pointer references a function that is used to search the buffer for the waveform of interest. The function does not necessarily need to process the buffer in the time domain; it merely returns a logical "yes" or "no" whether the desired biological event occurred within the current buffer. If, and only if, the first function returns true will the second function pointer be called. This function should define any action taken because of the biological event. If called, this function is processed directly after the first. The latency between the biological event and the required action is then proportional to system clock period. This latency is not programmable however; though this capability is certainly possible

and is discussed in the proceeding section. After the action function returns, a third function pointer is provided if the user needs to perform any additional processing that doesn't explicitly fit in the other two functions. This function is optional, though, and can be a blank stub. Program flow is shown in Figure 2.2.

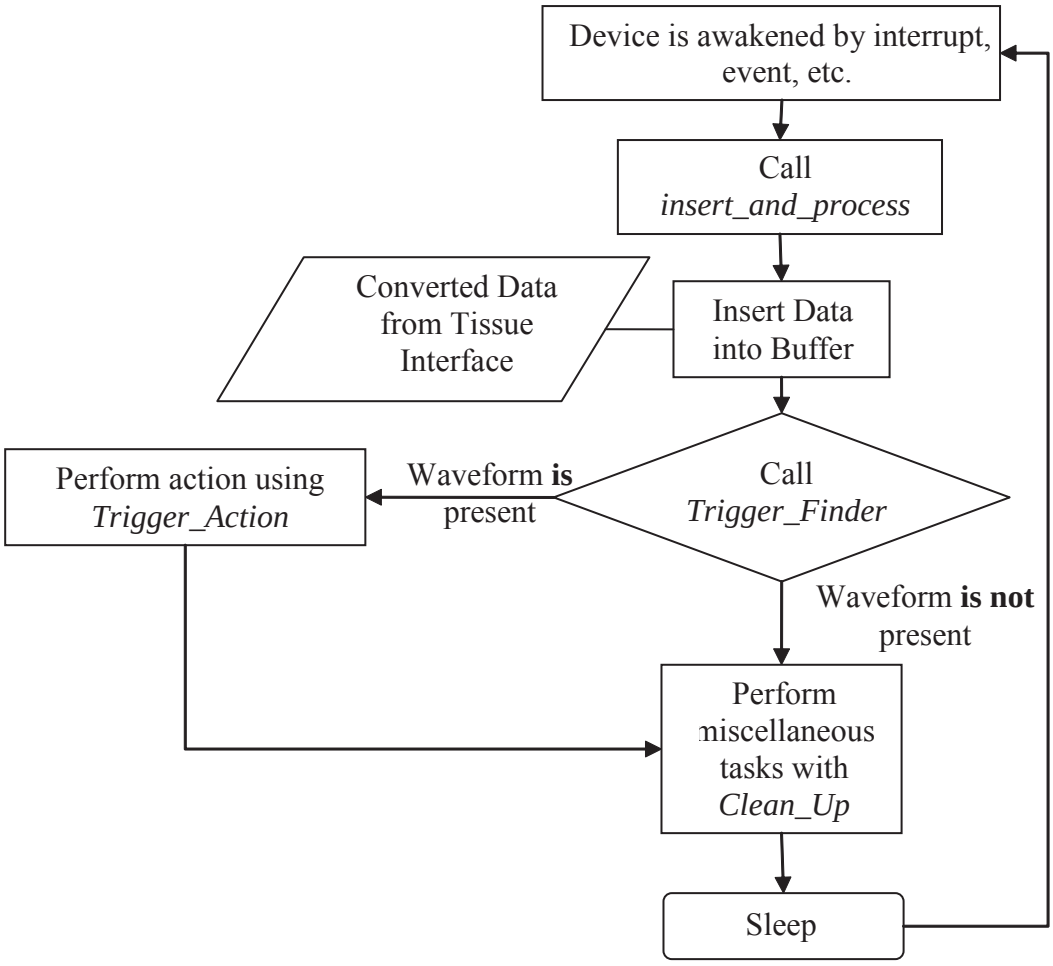

Figure 2.2. Channel Program Flow

Upon device wake up, the newly acquired data is inserted into the buffer. The buffer is then searched for the waveform of interest using *trigger_finder*. If found, *trigger_action* is called. Finally, *clean_up* is called, though that implementation is optional. After processing, the device returns to a low-power sleep state.

The channel structure also provides a third type of pointer, the void pointer. The void pointer can be used to reference any type of data. Its use is often discouraged for security reasons because arbitrary code can be inserted into otherwise normal behaving programs, but it provides useful functionality in the system presented here. The user can design a data structure of their choosing and bind it to the channel through this void pointer. Each of the three functions described above can then use the data in the structure. This usage will be more thoroughly discussed in the next section.

This library has been designed not only for easy usage, but also to adapt to a wide variety of brain-computer interface applications. It provides the basic functionality for any application and does not exclude the use of any detection algorithm. Because it is so general, the library alone is not suitable for any specific application. Researchers looking to use the library must build a layer atop the library implementing the specific algorithms for waveform detection and action. A widely-used application in the field currently is the detection of single-unit action potential waveforms.

2.1.2 Application for Automatic Action Potential Detection

Requirements

For numerous reasons, the detection and classification of single-unit action potential waveforms is useful in brain-computer interfaces. As the most discrete unit of information transmission in the nervous system, the spike trains of single-unit action potentials confer a high amount of information (Salinas and Sejnowski 2001). Detecting these pulses is useful for many scientific and clinical applications.

The small microprocessor used in this work provides no out-of-the-box capability for incoming or outgoing communication. As such, the device should operate autonomously with no need for user input. Especially as the channel count increases in brain-computer interfaces, the reliance on human operators to tune detection algorithms while detecting single

units online should decrease.

The voltage trace of an action potential recorded from a microelectrode displays a characteristic shape, first quantitatively studied by Hodgkin and Huxley (Hodgkin and Huxley 1945). The width in time of the action potential is around 1 ms and the peak-to-peak voltage can range from tens of microvolts to more than a millivolt depending on the electrode impedance and geometry to the cell (Joyner, Westerfield et al. 1980). Because even implanted microelectrodes can move in relation to neurons in the soft tissue of the brain, the shape of the recorded action potential from the same neuron can change over time and is therefore non-stationary.

A wide range of algorithms for detecting and classifying single-unit action potentials have been devised and each provides specific advantages and disadvantages. Each algorithm can ultimately be judged for its tradeoffs between computational complexity, accuracy, and to a lesser extent, latency in detection. The scientific pursuit will determine the optimal detection algorithm by placing a lower bound on required accuracy and device constraints will place an upper bound on the available computational resources.

The simplest algorithm is a voltage threshold such that any waveform crossing the threshold (which may be above or below the mean value of the signal) is considered an occurrence of an action potential. This algorithm is perhaps the easiest to compute and also the most susceptible to false positives. The recorded signal may contain many types of noise, common sources including power lines and subject movement. This noise can often wash out the extracellular action potential, thereby ruining the efficacy of the simple threshold. For recorded signals with relatively high signal-to-noise ratios (SNR), a simple threshold may perform adequately. However, the threshold must be initially set and because neural signals are not stationary, the threshold may need to be altered at some point during the detection. While quick to compute and easy to implement, the threshold algorithm rarely provides an optimal balance between accuracy and complexity.

Time-amplitude windows provide the next level of increasing accuracy and complexity. The window is often used in conjunction with a threshold to define voltage points the signal must exceed (a lower window) or pass below (a higher window). Buffer ranges can be open (with one point being the minimum or maximum of the dynamic A/D range) or closed (where both points are defined within the dynamic A/D range). A variation on the time-amplitude window is the template method. The algorithm uses a defined waveform to match incoming data. If the summed difference across every point falls below a threshold, an event is detected. The template method is essentially the time-amplitude window method with many windows of variable width (proportional to the tolerated error).

Both the time-amplitude window and template methods require initial values to begin detection, like the threshold method. There are numerous methods that improve detection accuracy but require increasing amounts of computational resources. Such methods include Principal-Component Analysis (PCA) and the non-linear energy operator. Online and offline detection methods are reviewed by Lewicki (Lewicki 1999). More complex algorithms increase accuracy at the expense of computational time. In high channel-count applications, these algorithms can only be used if there are available computational resources. Obviously, the available resources depend on the power constraints of the design.

Regardless of the spike detection algorithm, in order to maintain a portable library the algorithm must not depend on proprietary hardware features of a given microprocessor architecture. To this end, the algorithm should not require expensive computations such as digital signal processing routines or multiplication. It may sound odd to not rely on multiplication to implement spike detection algorithms, but small, low-power hardware architectures don't always include hardware blocks for multiplication. The C language requires implementation for this operation; therefore the compiler will link software routines to handle multiplication in such hardware. These software routines, while heavily optimized by the hardware manufacturer, are orders of magnitude slower to execute than hardware multipliers.

This work utilizes time-amplitude windows to detect neural waveforms. As discussed above, a spike detection algorithm in an embedded device should detect waveforms automatically. In this work, the algorithm to acquire the typical waveform shape and then begin detection is split into three parts, 1) setting the threshold, 2) defining the template, and 3) hunting.

Threshold

To place constraints in the automatic detection algorithm (and simplify the implementation), the initial assumption is that the extrema of the waveform lie outside the signal mean plus or minus some threshold. To determine when the waveform crosses one of these thresholds (whether above or below the mean in A/D space), a threshold must first be set. To determine the threshold placement, the algorithm immediately captures the first 2 ms worth of data points on the channel. The mean and peak-to-peak voltage is calculated using the following equations: $\mu = \frac{\Sigma}{N}$ where μ is the mean, $\sum$ denotes the summation of all points captured in 2 ms, N is the number of points; and $V_{pp} = \max(buffer) - \min(buffer)$. Standard deviation can be approximated by $\sigma = \frac{V_{pp}}{N}$, where N is 4≤N≤8 (Smith 1998). With the upper (mean plus standard deviation) and lower (mean minus standard deviation) thresholds set, a particular waveform can be found within the buffer whenever a particular point crosses the threshold. The more temporally older the critical point is set, the more of the entire waveform will be contained in the buffer. It should be noted that buffers should be more temporally wide than the signal of interest. For example, the buffer used in detecting single-unit action potentials should contain at least one millisecond's worth of data, the actual number of points being proportional to the sampling (insertion) rate.

Template

With the thresholds set, a new phase of signal processing begins. As a point in the buffer crosses the threshold, the entire buffer is copied into memory. This operation occurs sixteen times to create a matrix of waveforms. One dimension represents time while the other

represents instance of detected waveform. When this matrix is filled, the corresponding voltage (in A/D space) is averaged to create a template by which new spikes can be compared against. It may seem a contradiction that above multiplication was determined a flaw in a portable library and yet the average is computed here, which uses a multiplication. A digitally represented number can be divided or multiplied by a power of 2 by right-shifting or left-shifting the number a given amount of places, respectively. For example, a division by 16 is equal to a right-shift by 4 places ($2^4 = 16$) while a multiplication by 8 is equal to a left-shift by 3 places. The bit-shift is defined in every microprocessor instruction set and is extremely quick to compute. It is the otherwise portable multiplication routine across all hardware platforms, but is only useful for specific multiplicands or divisors.

Though this average is as temporally wide as the buffer and the template method could be used to detect spikes, the template method can be overly expensive. In this design, the minima and maxima are determined and two windows are created at the points. The windows have open ranges to decrease the computational complexity (passing the window only requires one comparison, not two).

Hunting

With the lower and upper windows set by the template phase, waveform detection is implemented by checking if the waveform currently held in the buffer passes both windows. This is essentially two integer comparisons and is extremely fast to compute. This algorithm may appear simple, but can be very accurate for even moderately isolated single units. To increase the number of channels that can concurrently be processed, the time spent computing each channel should be minimized as much as possible.

Implementation Details

The above discussion presents the general algorithm for automatic single-unit detection implemented in the current work. Specific details of the current implementation may elucidate how to adapt specific algorithms for future work.

Each phase of signal processing (thresholding, templating and hunting) is implemented as three functions designed to operate together with an external data structure. These three functions and user data pointer are discussed in the section above entitled "Channel Structure". In this implementation, the user data structure contains smaller, more specific data structures used solely in their specific signal processing scheme. There is a threshold data structure that tracks the mean and thresholds and this structure is written to during the threshold phase and then used in the template phase. The template data structure is defined by the large matrix of data points as well as an auxiliary pointer used to track the current point of insertion. The window group structure uses the template structure to place the upper and lower windows. The window group structure is then read when hunting. With the brief overview of how data is organized, we can examine the actual implementation more deeply.

When the device is powered on, numerous hardware pieces specific to the microcontroller are configured. Of most importance is the onboard A/D converter which is set to sample all channels every 125 μs for a sampling rate of 8 ksamples/s. The device is also placed into a low-power mode between channel processing and the next cycle. For more details on the specific hardware settings, see the "Microcontroller" subsection in Section 2.2. Once these settings are configured, the firmware then allocates all memory structures. After allocation, an internal timer used to drive the A/D converter begins cycling. The device now runs in the loop described previously until it is powered off.

Each channel begins with its function callbacks pointing to functions that implement the threshold phase. The *trigger_finder* function always returns logical true. The *trigger_action* function copies the newest datum point into a 16 point vector. When this vector has been filled, *clean_up* performs the following calculations. The mean and an approximation of the standard deviation are calculated as discussed previously. Thresholds are set above and below the mean by a constant factor of standard deviation, which is set at one. For recorded waveforms with large noise, this factor should be increased. At this point, the thresholds have been set, and the *clean_up* function alters the channel's function pointers to point to the next three functions implementing the template algorithm. This is summarized in the following

pseudo-code:

```
if (TRUE)
        Insert newest data in buffer to 2-ms wide array
if (array is filled)
        Compute mean, standard deviation
        Set lower and upper thresholds
        Set channel's function pointers to the template functions
```

The *trigger_finder* function in this phase returns true when the second point in the buffer exceeds the upper or falls below the lower threshold. When this occurs, the *trigger_action* function copies the buffer into the next row of an 8 x 16 matrix, each row representing an action potential, and each column representing a discrete time point. When this matrix is filled, the *clean_up* function averages the 16 points at each time point to create an 8 point average of the action potential. This average is searched for the maximum and minimum points, after which a window data structure is created containing these two time and voltage points. The channel's function pointers are then set to point to the block of functions implementing the hunting functions. This is summarized in the following pseudo-code:

```
if (buffer point crosses threshold)
        Copy buffer into matrix
if (matrix is filled)
        Average each index in the matrix
        Find the maximum and minimum points in the average
        Set windows at these time/voltage points
        Set channel's function pointers to the hunting functions
```

The hunting algorithm's *trigger_finder* uses the window data structure to search for action potentials. If the buffer passes through the two windows (only at the specific time points), then *trigger_action* flips a bit in an output data structure. This data structure implements counters to control delay between the action potential's detection and actual output, the width of the output pulses and a refractory period to stop multiple outputs for a single action potential. The following pseudo-code summarizes the hunting phase:

```
if (buffer passes through both windows)
        Enable output
if (output is enabled)
        Decrement delay counter
        if (delay counter times out)
                Do output action
                Decrement output counter
                if (output counter times out)
                        Turn off output
                        Decrement refractory counter
                                if (refractory counter times out)
                                        Reset all counters
                                        Disable output
```

Figure 2.3 graphically summarizes each phase of the algorithm. See Appendix A for a full listing of the source code.

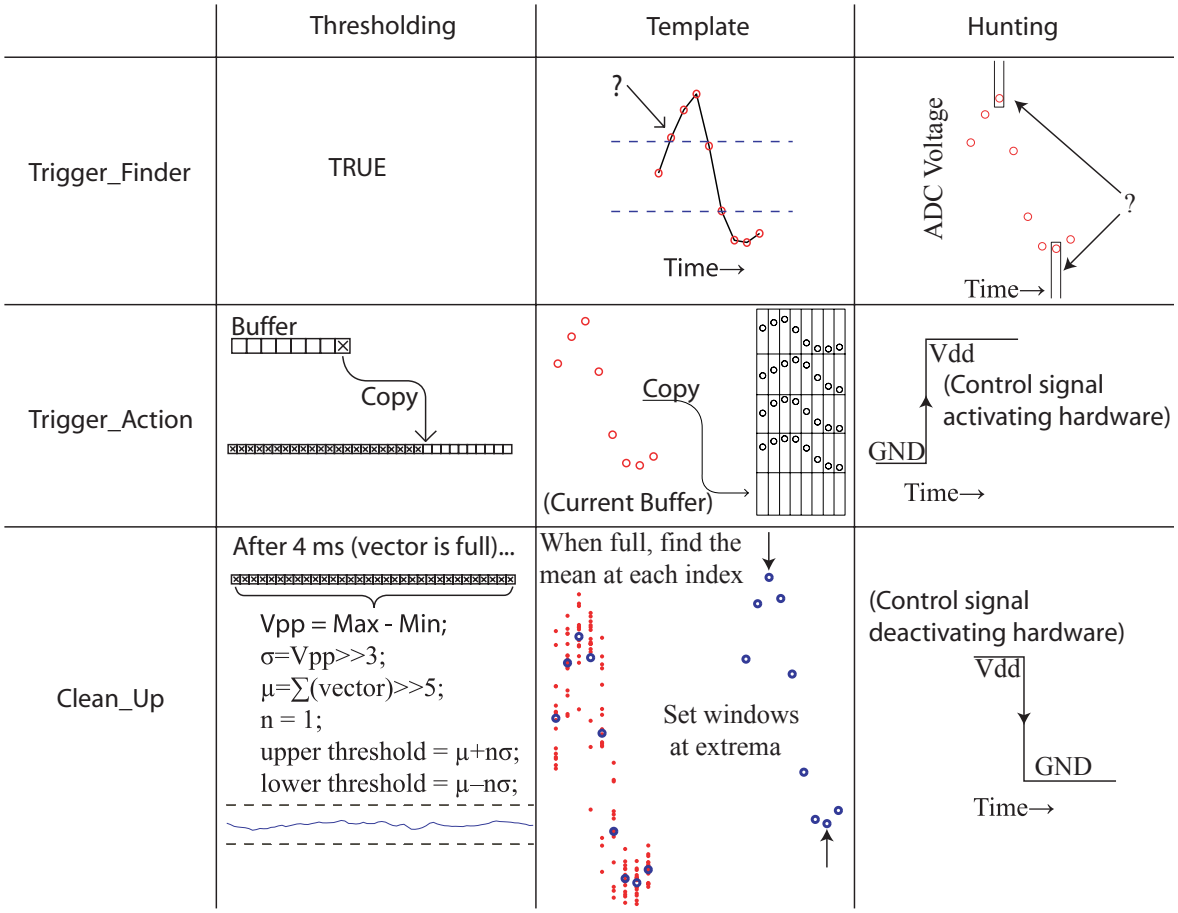

Figure 2.3. Phases of Single-unit Detection Algorithm

The role each function plays in each processing phase is graphically depicted. The algorithm is described in pseudo-code above, and the actual source code is in Appendix A.

2.2 Hardware

The firmware presented above is of little use in the physical world. It must be implemented in a microcontroller for any of the algorithms to actually run. This section describes the microcontroller, its auxiliary circuitry, and the other required hardware found on brain-computer interfaces.

2.2.1 Signal Conditioning

Requirement

The signals acquired from the brain are extremely small in both amplitude and power (Hodgkin and Huxley 1945). To effectively use the dynamic range of the A/D converter, the biological signals should be amplified. Also, the signal's power should be increased; otherwise intermittent EMF signals will cause distortion, especially when the signal is transported over relatively long distances through cabling, as would be expected when transporting the signal from the brain to a backpack, where the device is meant to be housed.

Bandpass Filter

To accurately convert an analog signal to the digital domain through an A/D converter, care must be taken to ensure the analog signal does not contain frequencies greater than half the sampling rate of the A/D converter. The optimal filter would possess a rectangular response in the frequency domain, perfectly shunting all signals falling outside of the pass band. In practice, this is not possible for numerous reasons. Digital filters can closely approximate perfect filters (nearly to the float error the hardware platform) but can be prohibitively expensive in time and energy. Analog filters cannot perform with such accuracy as digital filters without requiring large physical space, but are more energy efficient.

If an action potential is transformed into the frequency domain, the majority of the signal

power lies between 300 and 3000 Hz (Pattle 1971). To reduce the unwanted frequencies (which represent sources of noise) and satisfy the Nyquist Theorem, a 4-pole Butterworth band pass filter was designed using Sallen-Key topology.

There exist a few different filter designs, each with their advantages and disadvantages. The Chebyshev filter provides a steeper roll-off (the transition from the pass band to the stop band), though there can be significant ripple in the pass band. These filters also do not provide linear phase responses, which can be significant when reconstructing signals. Bessel filters provide optimally linear phase responses at the expense of a very slow roll-off. Elliptic filters provide the steepest roll-off but there is large rippling in both the pass and stop bands. Butterworth filters provide a balance between no pass band rippling, but with slower roll-off performance (Butterworth 1930).

The Sallen-Key filter topology is simple and particularly suited for low-cost, power-efficient operational amplifiers (Karki 2002). Each two-pole stage requires four passive components and one active operational amplifier. Putting these stages in series increases the poles for the filter and thus increases the roll-off. The low-pass four-pole filter is designed with a cutoff frequency at 3000 Hz, while the high-pass four-pole filter's cutoff frequency is designed at 300 Hz. The incoming signal enters the highpass stage first so amplified low frequencies do not saturate the operational amplifier. The filters with component values are shown in Figure 2.4.

Amplification

Implicit in Figure 2.4 is the circuit gain. Action potentials are rarely larger than +/−1 mV in amplitude. As such, a gain of 1500 has been designed into the component values so +/− 1 mV signals will fill the entire range of the A/D converter. In practice, signals will not approach the 1 mV amplitude because relatively low-impedance electrodes are used.

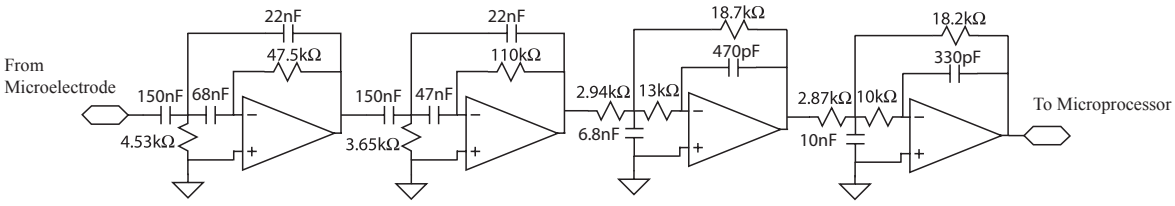

Figure 2.4. 4-Pole Butterworth Sallen-Key Bandpass Amplifier/Filter

Circuit for the described amplifier/filter used to condition incoming neural signals before A/D conversion. The four operational amplifiers are implemented in one 16-pin package, the Texas Instruments TLV2375 low-power operational amplifier.

2.2.2 Stimulation

This engineering project has been developed in parallel with a scientific hypothesis that artificially-induced spiking can induce long term plasticity in biologically relevant neural networks. As such, a small, efficient stimulator is needed on board to create stimulation currents to pass to neurons.

Requirement

The most widely-used modern technology to stimulate nerve cells is electrical stimulation. Bio-electrically speaking, nerve cells depolarize when current is drawn out of the cell membrane and into the microelectrode (a negative current relative to the electrode). So as to not deposit metal in tissue, an equal and opposite positive pulse should accompany the negative pulse to reverse the chemical reaction occurring at the electrode-tissue interface (Brummer and Turner 1975). This waveform is called the biphasic pulse. A stimulator must thus be able to swing in both directions to produce the required waveform.

Current injection can occur with either a voltage source or current source. Ohm's Law describes the relationship between current and voltage as such: $V = I*Z$, where V is the voltage, I is the current flow, and Z is the impedance across the point that voltage is measured. On a bench top, with a simple resistor providing the impedance, this relationship is simple. To the stimulator, the impedance is measured between the positive and negative output terminals. In a biological environment, this includes the microelectrode and tissue. The microelectrode impedance will not change significantly through an experiment, but tissue impedance can. Therefore, at the beginning of an experiment a certain voltage may provide adequate current injection to stimulate nerve cells, but it is not guaranteed to always be the case. The more robust method is to provide a known quantity of current at all times, regardless of the impedance (Stoney, Thompson et al. 1968).

Improved Howland Current Pump

When the load is single-ended and bipolar current is required, the Improved Howland Current Pump is a proven, simple design. The circuit can be found in Figure 2.5. The resistors form a bridge about which the operational amplifier is forced to push current into the load (Pease 2008). The current output through the load is given by

$$I_{load} = \frac{V_{IN1} - V_{IN2}}{R_{13}} \text{ when } \frac{R_{14}}{R_{15}} = \frac{R_{11}}{(R_{12} + R_{13})}.$$

To achieve the desired biphasic waveform, the input pins are tied to general purpose I/O pins on the microcontroller. Thus, when one pin is held at high voltage and the other low, one polarity will be generated. When the pins are flipped in voltage, the opposite polarity will be generated, and the current pump will create the opposite current.

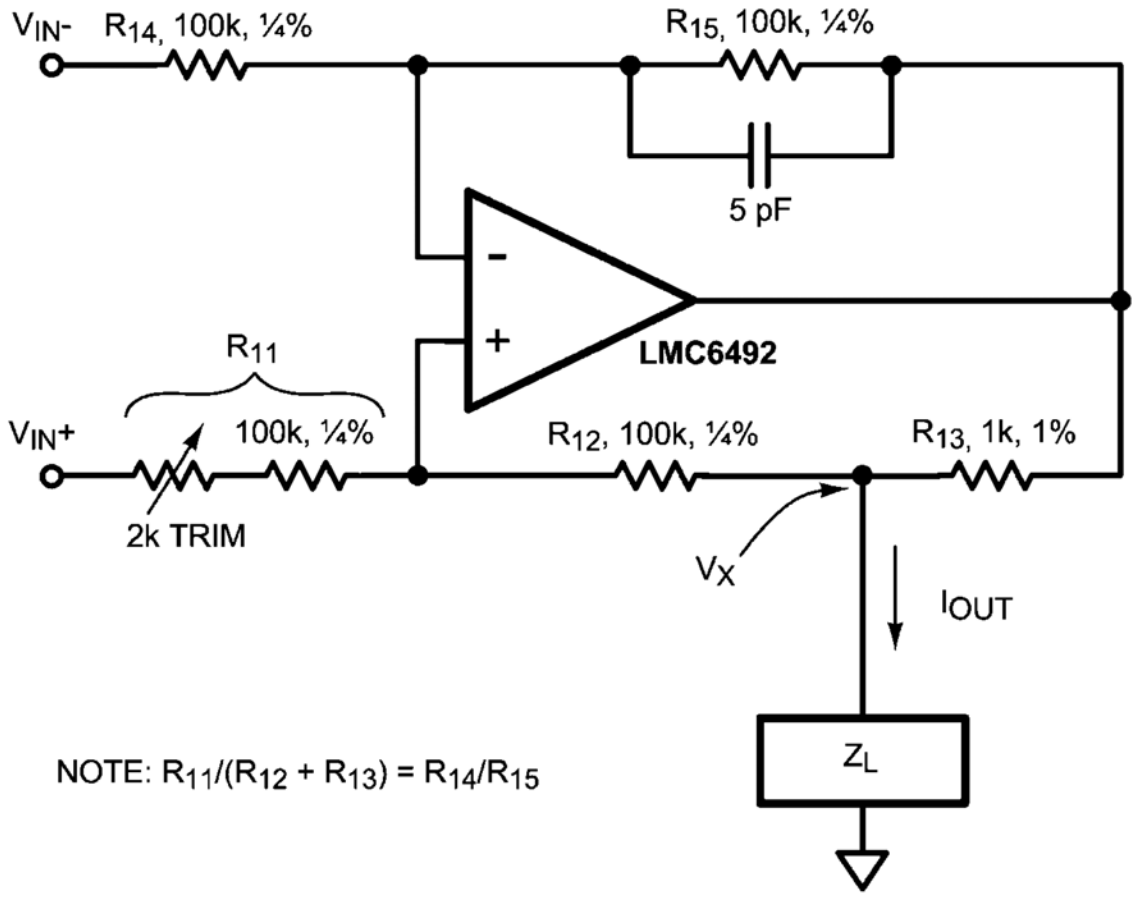

Figure 2.5. Improved Howland Current pump

From (Pease 2008). These component values are generic and must be calculated for a particular application, but this is the general circuit used to deliver stimulation pulses in this design.

Voltage Boost

Of course, Ohm's Law still applies for the generation of current. Therefore, the operational amplifier will need to be biased with large enough positive and negative voltages to allow it to generate the desired current. By Ohm's Law, to produce 30 μA through a 500 kΩ microelectrode, the operational amplifier will require +/− 15 volts. Due to the board's size constraints, batteries supplying this voltage cannot be included in the design. The voltage must be generated through a DC-DC boost converter. These circuits can produce voltages higher than their input by manipulating power, which must be conserved. By storing a small voltage with lots of current, the boost converter uses the excess current to increase voltage. They thus transform small voltage, high current loads into high voltage, low current loads of equal power.

In this circuit, the boost converter needs to activate only just before stimulation, and the chosen boost converter IC provides a disable pin, that when drawn low, will disable the output. This is particularly useful to save battery life as all boost converters use a relatively high amount of current and can drain a small battery quickly. The circuit diagram for the boost converter and improved Howland current pump is shown in Figure 2.6.

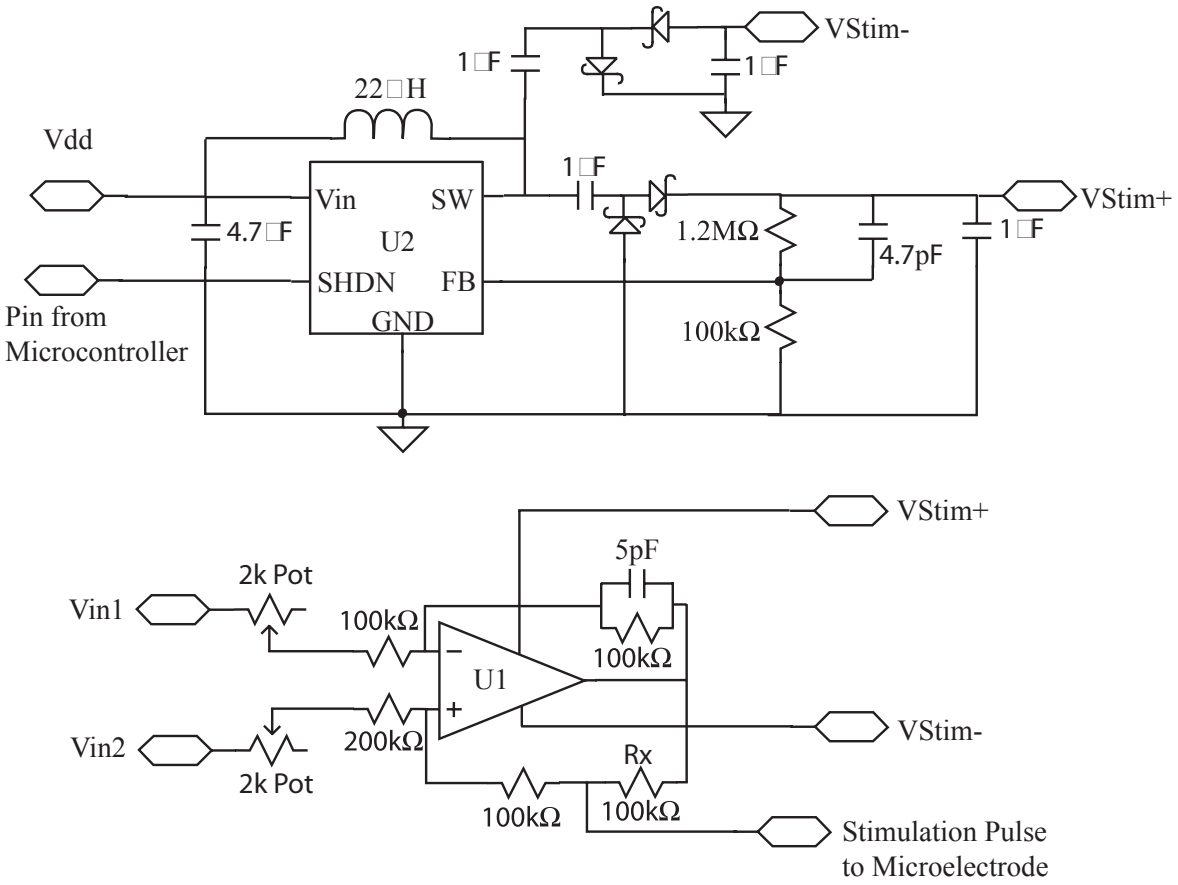

Figure 2.6. Voltage Boost and Improved Howland Current Pump.

The implemented Improved Howland Current Pump and voltage boost circuit. The voltage boost IC is the Linear Technology LT1615 and the operational amplifier used in the current pump is the Texas Instrument TLV7302.

2.2.3 Power Supply

A fully tether-less BCI obviously cannot include a power line to the wall. Some sort of power source is necessary to provide energy to the microcontroller and related circuitry. This power source should provide a balance between size and operating life for the BCI.

Requirements

Modern electronics cannot be powered simply from a battery. While a battery provides a good source of portable energy, the output voltage can be extremely noisy. Digital electronics require very stable supply voltages to operate accurately. Therefore, it's necessary to employ a voltage regulator to create a clean voltage source. Voltage regulators come in two types, (1) linear and (2) switching. Linear regulators produce extremely clean voltage outputs, but are inefficient. Switching regulators consume very little power but the output voltage source can be somewhat noisier. Linear regulators also require less passive components, which can be useful for smaller designs.

Power Source

The design is driven by a ½ AA battery (Tadiran Model TLL-5902). This power source provides 3.6V of unregulated voltage with a maximum continuous drain of 50 mA. It can provide 100 mA for brief periods of time. This battery is ideal because it provides a more than twice voltage at half the size of a normal AA battery. Not only can the board be smaller, but the overall weight will be much smaller. This particular battery contains about 1100 mA-H of power.

Voltage Regulator(s)

Two voltage regulators are used in this design. The first is the main voltage regulator that provides 3.0V of regulated power to the digital and analog ICs. The second voltage regulator provides 1.5V of regulated power that is used to ground the animal. Because the animal is grounded at this midpoint in the supply voltage, the signals coming from its brain are biased halfway in the A/D dynamic range. The two regulators used are the TPS79030 and the

TPS79015. The power supply circuit is shown below in Figure 2.7.

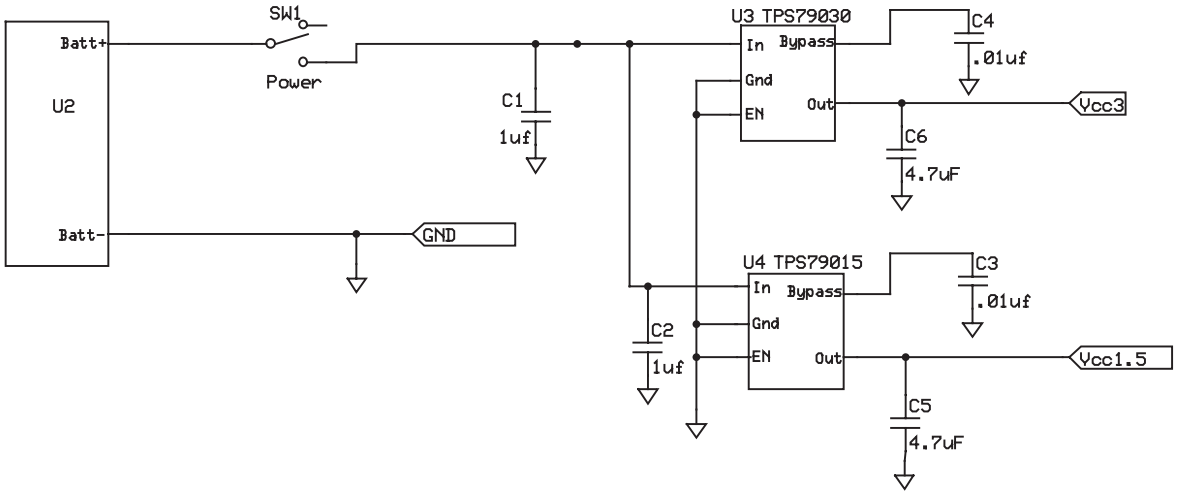

Figure 2.7. Power Regulator Circuit

The circuit defining the battery and voltage regulators.

2.2.4 Headstage

Requirements

Head stages are important pieces of hardware in neural data acquisition systems that help combat noise issues by amplifying and/or buffering the signal close to its source. The microcontroller used in our design is intended to be headstage-independent. To employ the full dynamic range of the A/D converter, voltage signals from the headstage should not exceed V_{dd} or fall below GND and be biased around half V_{dd}. Triangle Biosystems (Durham, NC, USA) produces popular electrode/headstage systems suitable for implanted designs for small animals (Hampson, Collins et al. 2009; Rolston, Gross et al. 2009).

2.2.5 Microcontroller

Arguably the most important piece of hardware in this design, the microcontroller performs all relevant processing tasks. As semiconductor manufacturing processes shrink, many hardware blocks such as flash memory, peripheral communications, and A/D converters can be integrated in a single IC, providing a complete system. Modern microcontrollers provide an efficient, low-power solution to many embedded computing tasks in which a full operating system or microprocessor is not required. Power consumption, CPU efficiency, size, and the programming/debugging environment are key design factors when choosing microcontrollers. I have chosen Texas Instruments MSP430F5438 microcontroller, and subsequent discussion will focus on its capabilities.

Power Consumption

In battery-powered applications such as this design, the most important characteristic is power usage. The power consumption of the microcontroller is dependent on two factors, the 1) manufacturing process at which it is implemented, and 2) the usage. As users of the microcontroller, the first point is out of the scope of our capabilities. The usage, however, is completely within our control. The MSP430 platform provides a complex power-management unit (PMU) that provides varying degrees of power consumption modes. The

most consuming mode is termed the active mode, in which all clocks are running, the CPU is actively processing data, and all peripheral hardware blocks are capable of running. At the opposite spectrum is a sleep mode in which the only power consumed is used to keep memory in the correct state and run one clock.

Independent of the power mode, the operating frequency can be modulated. Power consumption is obviously proportional to clock speed, i.e. the microcontroller will consume more power the faster it runs. To conserve power, it may seem obvious to decrease clock speed, but such is not the case in this situation. After all channels have been processed for a single loop iteration, the device is placed into the sleep mode. The device consumes such little power in this mode that it is actually beneficial to operate the device at its maximum clock speed when processing the channels. By reducing the time spent actively processing (and therefore using the power-hungry CPU) and maximizing time spent in sleep mode, power usage can be minimized.

CPU Efficiency

The MSP430F5438 CPU is a 16-bit, reduced instruction set computer (RISC) designed with a von Neumann architecture. The CPU only implements 27 total instructions which allows for more efficient execution. The von Neumann architecture can pose a bottleneck because both data and instructions transferred to the CPU through the same physical bus. The pipeline between CPU and memory must therefore move both the instruction and operand before the CPU can compute the result. The Harvard architecture provides parallel pathways for both data and instructions and is found in competing microcontrollers (Smith 1998).

On-Board Peripherals

The MSP430F5438 provides many useful peripherals for this application. The 12-bit A/D converter (ADC) employs a successive approximation algorithm to convert analog waveforms to discrete values. When triggered to sample and convert, the input voltage is locked (termed 'sample and hold'). For the next 12 clock cycles of the ADC, the signal is compared against a

successively halved value to implement a binary search that eventually produces a 12-bit discrete value of the analog voltage. A reference voltage is required in this algorithm to normalize the input. Assuming the input signal and reference voltage are measured against the same ground, the ADC conversion can be modeled as the following: $Out = round(\frac{V_{in}}{V_{ref}} \cdot 2^N)$ where N is equal to the bit depth of the ADC.

The quantization error of the MSP430F5438s SAR ADC is half the least significant bit which can be found with the following equation: $Q_{error} = \frac{V_{ref}}{2^N}$. Therefore, the least significant bit represents ~700 μV and the largest quantization error is less than 400 μV.

The ADC circuitry in the MSP430F5438 can maximally convert 200 ksamples/s (Texas Instruments 2009) but is set to sample every 125 μs in this application (for a rate of 8 ksamples/s). A hardware timer is used to generate this clock source. The clock runs in parallel to any processing in the CPU and is not stopped during the low-power sleep state. This clock wakes up the ADC circuitry when it reaches zero. The ADC circuitry converts samples for all channels and generates its associated interrupt. It is in the ADC interrupt that processing takes place. When the interrupt finishes, i.e. all the channels have been processed, the microcontroller returns to the sleep state.

The MSP430F5438 also contains numerous hardware capabilities for peripheral communication. Interfacing the microcontroller with some sort of data transmission device could be accomplished with any number of communication interfaces such as SPI, I^2C, or UART. Areas of improvement and future development are discussed in Chapter 4.

Programming

The entire source for this project has been written in C for a few reasons. First, C is the most prolific language and there exists a C compiler for every microcontroller platform. Thus, because the library and application have been written in C, this source is extremely portable to other platforms. There also exist C compilers for normal desktop computers, for instance

the freely available GNU Compiler Collection (http://gcc.gnu.org). A majority of the development of this work was actually done by compiling the code for a Windows machine, and setting the program to parse a text file of simulated digital values for a train of action potentials. The ability to cross-compile this code readily facilitates new algorithm development for future implementations. Of course, some development must be tested in the actual microcontroller, for instance the optimal settings for hardware blocks. As those settings are tuned for a particular microcontroller, though, there will be little need to change them.

When using source code written in a high-level language such as C, the chosen compiler can have drastic effects on the performance of the code. The compiler transforms human-readable source code to machine code, a binary string representing all data and instructions. The more proven and reliable the compiler, the more efficient and smaller the generated machine code will be. Texas Instruments provides a toolset for the MSP430 platform including the compiler, assembler, archiver, and linker. This toolset was used to generate code for the microcontroller while the GNU Compiler Collection was used to generate x86 code running on a Windows personal computer.

Debugging

Debugging source code in full-fledged computing environment such as Windows can be difficult. Hence, debugging running code on a microcontroller that possesses orders of magnitude less I/O and communication resources can present lots of challenges. Luckily, Texas Instruments makes available their MSP430 Flash Emulation Tool USB Debugging Interface. This small device connects to a personal computer through a USB port and connects the integrated development environment (IDE) to the microcontroller running the program. With this combination, compiled code can be programmed onto the microcontroller and be verified to work. Breakpoints can be set and individual registers and memory can be monitored during runtime. Without this toolset, debugging would be much more difficult.

2.3 Discussion

I have outlined the design and implementation choices one faces when designing a small, low-power, extensible brain-computer interface. To be of use and test my design, I also created an algorithm to automatically detect single unit action potentials. This is but one application in the growing field of brain-computer interfaces. It is my belief that the library discussed above provides everything a new BCI designer would need to begin development of a new device using an off-the-shelf microcontroller.

There is nothing explicitly novel about any of the hardware designs I've presented. They are included to provide a complete system and should be tuned for each particular application. The novel approach of this work is to generalize the handling of neural data, allow for a wide variety of processing techniques across all channels and decrease the cost of development for new brain-computer interface devices.

Chapter 3 Testing and Validation

3.1 In-Silico

To test the basic viability of the code, the source was compiled on a Windows desktop computer using the Cygwin package (http://www.cygwin.com). This package provides a substantial Linux API, which is helpful in compiling C applications on Windows without having to use Microsoft's development tools.

3.1.1 Testing Method

About one minute's worth of previously recorded single unit action potentials were scaled and manipulated in MATLAB (Mathworks, Inc.) to reflect actual values from the A/D converter (i.e. they were scaled to the A/D range and biased) and written to a text file. The desktop application (see PNCS_App_desk.c in Appendix A) was configured to read this text file, process the data as described in Chapter 2 and output various debugging information to console. The program also writes to an output file with a "0" if there was no spike detected and a "1" if a spike was detected. Both text files were loaded into MATLAB and examined.

3.1.2 Expectations and Actual Results

The printed debugging information was captured and is below.

```
Setting baseline...
Baseline set.
Baseline:
Mean: 2040     SD:259
Upper: 2558    Lower: 1522
Learning...
Learning complete.
Full Templates:
2128 2407 3037 3555 2901 2008 1379 966 721 682 872 1155 1462 1751 1958 2113
2049 2309 2834 3454 3043 2228 1634 1283 1058 911 877 947 1104 1351 1644 1913
2162 2432 3029 3503 2775 1887 1331 906 698 768 988 1291 1569 1799 2016 2187
2086 2322 2812 3488 3135 2226 1520 990 694 595 683 953 1284 1609 1908 2114
2103 2370 2889 3427 2927 2151 1587 1228 1013 904 902 986 1192 1454 1687 1889
2117 2396 2975 3458 2902 2144 1595 1266 1092 1001 980 1083 1308 1525 1730 1913
2174 2446 3001 3407 2818 2025 1344 899 665 632 765 1049 1373 1674 1935 2132
2070 2276 2580 3209 3289 2571 1893 1356 1025 811 748 853 1073 1349 1620 1870
2155 2481 3127 3469 2728 1898 1303 877 657 725 972 1282 1586 1838 2046 2169
2133 2363 2810 3417 3108 2313 1704 1281 954    798 798 904 1139 1442 1688 1862
```

```
2189 2467 2870 3427 3034 2244 1610 1139 852 692 700 857 1109 1410 1725 1988
2077 2333 2696 3225 3062 2403 1843 1466 1234 1063 1012 1067 1173 1325 1544 1733
2119 2395 2934 3393 2843 1996 1317 862 618 589 711 961 1300 1645 1916 2141
2078 2287 2586 3218 3303 2580 1884 1293 904 731 716 836 1108 1401 1678 1930
2191 2459 2970 3400 2855 2039 1349 864 590 526 695 979 1310 1640 1952 2193
2056 2224 2630 3321 3259 2413 1686 1182 838 650 691 915 1248 1585 1862 2052
Mean template:
2117 2372 2861 3398 2998 2195 1561 1116 850 754 819 1007 1271 1549 1806 2012
Max Window:Index:      3       Value:  3398    Direction:     1
Min Window:Index:      9       Value:  754     Direction:     0
Hunting...
Hunting complete.
Baseline run time: 0.000000 sec
Learn run time: 0.031000 sec
Hunt run time: 2.063000 sec
```

The thresholds and average templates are all automatically set, and hunting through 60 s of data takes about 2 s. Obviously, the program works and is fast. Figure 3.1 reveals how well the program detects action potentials. The outputted "1"s have been scaled to fit on the graph. The initial few action potentials are not detected because they were used for the template phase.

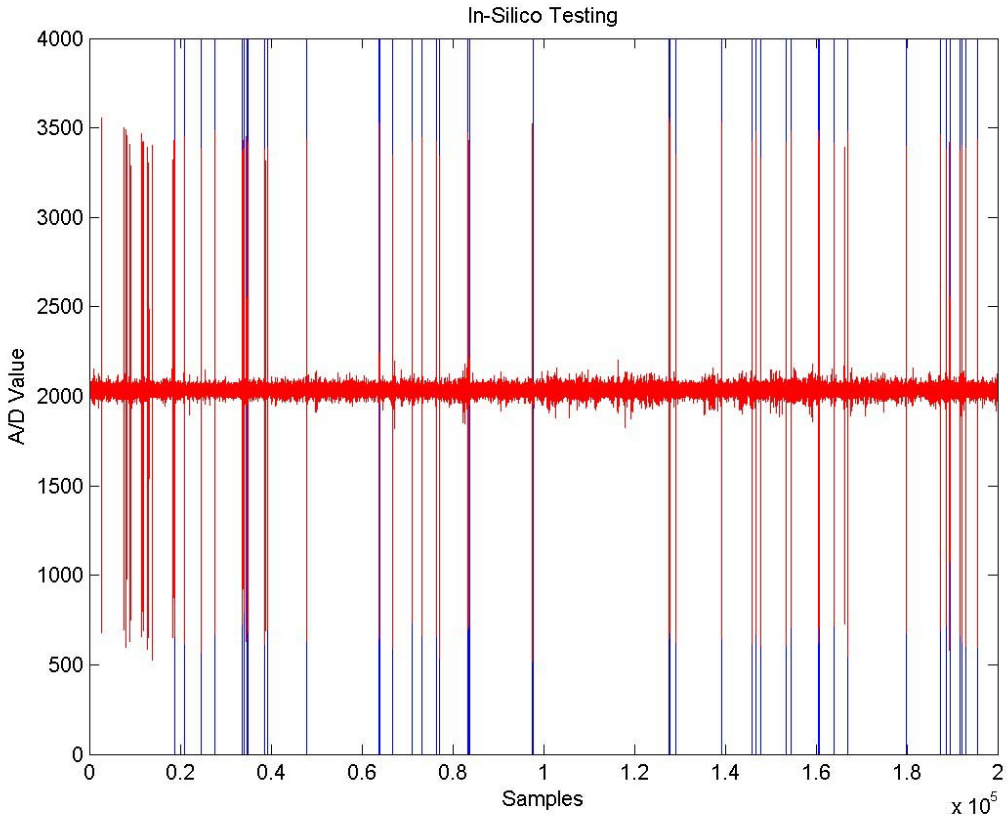

Figure 3.1. *In Silico* Testing of Library and Desktop Application

The simulated neural waveform (red) imposed on pulses (blue) output by the application into a text file. The early action potentials were missed because those waveforms were used for templating.

3.2 Benchtop Simulation

3.2.1 Testing Method

To test the efficacy of the firmware/hardware combination in detecting single-unit activity, we first simulated extracellular action potential waveforms through a National Instruments data acquisition board (National Instruments PCI-6229) controlled by MATLAB (Mathworks, Inc.). Using single-unit data recorded in our laboratory from awake common marmosets (*Callithrix jacchus)*, we delivered the waveforms through a hardware breakout box (National Instruments BNC-2090A) and connected the waveforms to the A/D pins on the microcontroller. At the time, the microcontroller was housed in a breakout board connected to a debugger (MSPFET430UIF, Texas Instruments). The National Instrument board also recorded pulses from the microcontroller during each playback/recording session. A pulse from the microcontroller signified the detection of an action potential. In the current implementation, four channels of single-unit data can readily be detected in real time. All channels process data independently and can detect action potentials of differing shape and size. A 10 s session of four-channel recording is displayed in Figure 3.2.

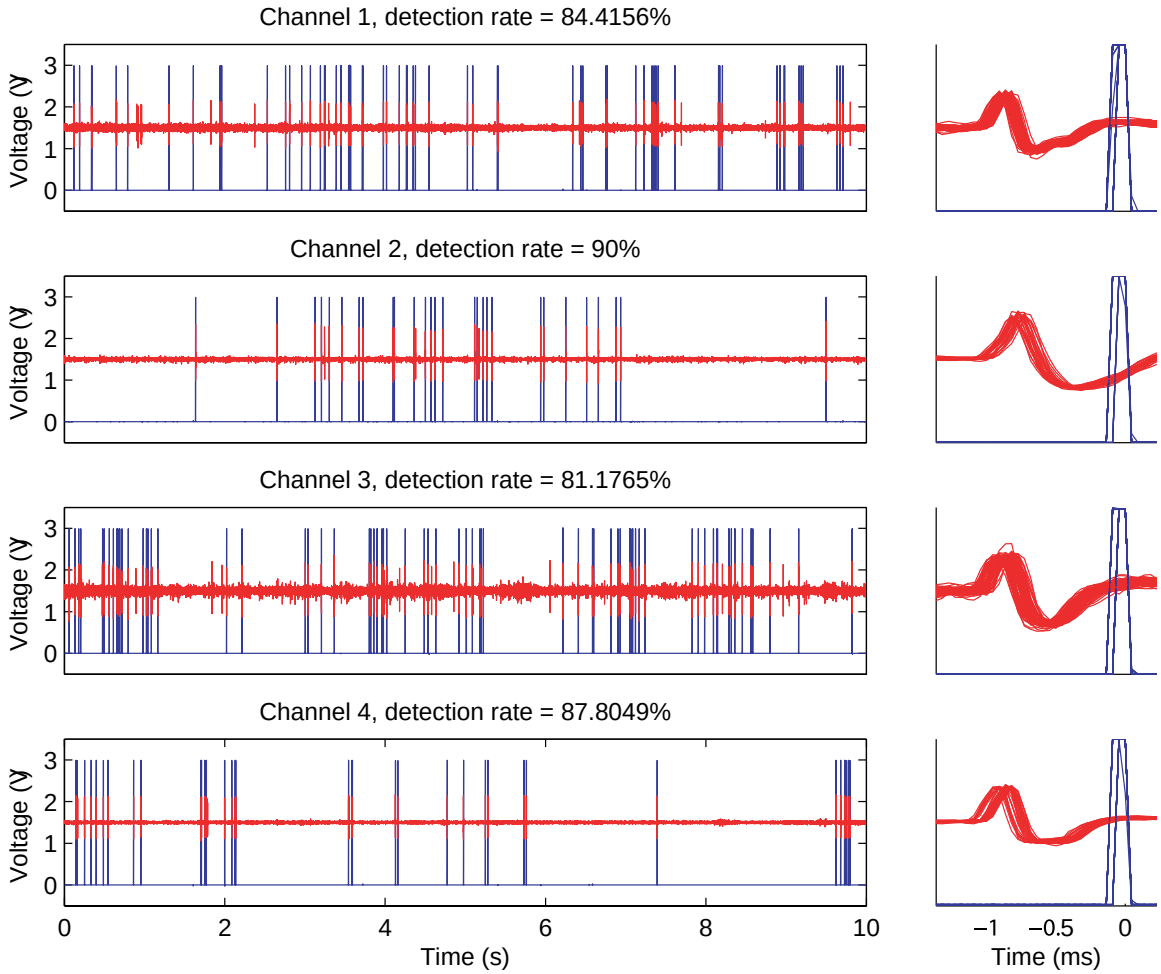

Figure 3.2. Four Channels of Simultaneously Processed Data

The microcontroller can accurately decode four channels of neural waveforms simultaneously. The waveforms (red) was prerecorded from our primate model, the common marmoset (*Callithrix jacchus)* and delivered to the microcontroller through the analog output of a National Instruments Data Acquisition Card. The microcontroller's response (blue) represents action potential detection. The waveforms and response were recorded on the same National Instruments card. All channels were sampled by the National Instruments card at 20.8 ksamples/s. The plots on the right depict the latency between action potential and detection, which is less than 1 ms for each channel.

3.2.2 Expectations and Actual Results

The automatic action-potential algorithm detailed earlier found around 85% of the action potentials on a given channel in our initial tests for a miss rate of ~15% and a false positive rate of 0%. A possible solution to increase detection of true spikes (at the risk of increasing the number of false positives) is to relax the constraints on the time-amplitude windows, i.e. decrease the upper window amplitude and increase the lower window amplitude. Ultimately, the BCI application will determine the required accuracy in action-potential detection.

3.3 Discussion

Of course, if the shape of the action potential is known *a priori* at compile time, appropriate windows can be hard coded. I chose to implement a more flexible approach due to the general instability of single-unit recording. As single units are invariably lost from day to day during an experiment, simply resetting the microcontroller will restart the threshold, template, and hunting sequences. This approach lessens the burden on the operator to continually place time-amplitude windows to optimally discriminate action potentials.

Chapter 4 Conclusion

This thesis presents the design and implementation of a generalized firmware platform upon which brain-computer interfaces (BCIs) can be built using off-the-shelf microcontrollers. Current implantable BCI designs generally lack extensibility and customization, thus limiting their usefulness (Mojarradi, Binkely et al. 2003; Harrison, Kier et al. 2009; Raghunathan, Gupta et al. 2009; Rizk, Bossetti et al. 2009). The basic firmware platform presented here can reduce design and development time of future BCIs by abstracting data storage and program flow. This abstraction allows researchers to focus development efforts upon their specific application, namely waveform detection and the action to be taken upon detection.

4.1 Advantages/Disadvantages

A BCI implemented with a general-purpose microcontroller will maintain certain advantages over BCIs implemented with ASICs. Firmware written in the industry standard C language reduces programming difficulty for new users and developers. Comparatively, designing ASIC circuits is time-consuming, expensive, and difficult for those without extensive training. Many microcontroller makers provide rich integrated development environments (IDEs) to assist in program development and debugging. Also, many companies provide development boards for their microcontrollers, accelerating application-specific development for these general devices. Due to the portability of the C language, specific BCI development can begin even before a microcontroller has been selected. The firmware can be compiled on any operating system and tested as a standalone program, as shown above.

The re-programmability of the microcontroller adds immense value to this specific brain-computer interface (An, Park et al. 2007). As a BCI experiment proceeds, external factors may require changes to existing functionality or call for a completely new functionality. If the changes required are within the capability of the firmware, then this new functionality can be implemented in the same implant quickly. Once the new functionality is integrated into the existing firmware, the device can then be reprogrammed in a matter of seconds and the experiment can continue. Currently published BCI designs do not provide the ability to add features on this time scale (Mavoori, Jackson et al. 2005; Ye, Wang et al. 2008; Rizk, Bossetti et al. 2009).

Applications such as artificially-induced plasticity (Jackson, Mavoori et al. 2006) require low latency between the actual action potential, its detection and the action taken. The common practice of spike detection and sorting in a remote computer is ill-suited for closed-loop application. There is an inherent time delay in wireless transmission of data to the remote computer, processing, and then the reception of control signals—especially when transmission bandwidth is limited in low-power devices and/or with high channel counts (Bossetti, Carmena et al. 2004). A 50 ms delay may be tolerable in some applications, but a generalized platform should set the lower bound of response time much lower to increase usefulness.

When input/output is not available or too computationally expensive, intelligent on-board algorithms are needed for the automatic detection of action potentials. These algorithms must trade off detection latency with hit rate (or, equivalently, miss rate) and false positive rate. A completely automated initial algorithm is presented here that displays a miss rate of about 15% and a false positive rate of 0% for well-isolated single units, all with a detection latency of about a millisecond from spike onset. Shorter detection latency could be achieved at the expense of false positive rate. The requirements of the application and the quality of signals extracted from the tissue interface will ultimately determine the best compromise for these parameters. Because of the ease of reprogramming, this firmware platform can readily accommodate a wide variety of applications.

The automatic spike detection algorithm presented here was designed to explore the effectiveness of using general purpose microcontrollers in BCI designs. Depending on the nature of the signal from implanted microelectrodes, this algorithm may be severely overkill (i.e. the signal-to-noise ratio (SNR) is high and a simple threshold would be effective) or woefully inadequate (i.e. the amplitude of the noise approaches that of the action potential or there are multiple single-unit action potentials on the microelectrode). Nonetheless, this algorithm demonstrates the flexibility and extensibility of the firmware design. It should also be apparent that this design is not limited to only detecting single-unit action potentials. If a signal can be translated to the digital domain and an algorithm can be devised to detect the signal, then this firmware library can accommodate such a sensor design.

ASIC-based BCI designs will always hold some advantages over general-purpose BCI designs. Off-the-shelf general purpose microprocessors are implemented in industry-standard packages without the capability of hardware customization. ASIC manufacturers can print any size ASIC, and therefore custom ASIC BCIs can be very small. The entire device presented in (Harrison, Kier et al. 2009)

measures 5.4 by 4.7 mm^2 compared to the MSP430F5438 alone at 13 x 13 mm^2. The size issue limits microcontroller-based designs to usage in moderately sized mammals (rats, marmosets, and larger). ASIC designs can also consume much less power than microcontroller-based designs. General purpose microprocessors include extraneous hardware blocks that are often not needed in BCIs, thus representing wasted power and space. Modern central processing units found in today's microcontrollers and microprocessors are highly optimized but are still designed to implement an instruction set. ASICs found in BCIs have no such requirements and can therefore use a reduced transistor count.

4.2 Scalability

The portability of the presented firmware makes possible not only a great many applications, but also its use in widely varying devices. In the context of low-powered BCIs, the firmware can be targeted at small microcontrollers. The MSP430 platform is specifically designed for ultra-low power usage. If the power constraints in the BCI design are relaxed, then a more efficient, high-power microcontroller could be employed. For instance, the microcontrollers used in current smart phones not only include more efficient CPUs but also integrated graphics processors. These graphic processors are extremely optimized for matrix and vector operations and could be employed for powerful spike detection algorithms. Such algorithms exist but have only been implemented in the larger graphics processors found in personal computers (Wilson and Williams 2009).

One could also imagine a rack-mounted design employing a higher power microprocessor capable of monitoring many channels in real time. The MSP430F5438 used in this design operates at 18 MHz. Even assuming no gain in processing efficiency, a two order-of-magnitude increase in clock speed would allow for more channels than many commercial recording systems. Rack-mount systems could also include high-density storage for action-potential shape and times as well as serial (USB, Firewire) or Ethernet communication to desktop PCs.

This firmware could also be used as the back end for a software application if targeted against a modern operating system and run on a desktop PC. Operating systems abstract the hardware layer and increase processing overhead, but increase the number of machines that can run the software. Desktop PCs do not have the memory constraints of embedded microcontrollers, but such a program would still benefit from a lightweight, efficient back end for memory storage. Of course, this back end

would be portable across Windows, Macintosh, and Linux OS platforms.

It's not out of the realm of possibility to think about using actual smart phones as the processing hardware in BCI designs. They may be too large to include in a backpack design, but for tethered designs in which implanted electrodes are chronically connected to the BCI through the cage, this could be very feasible. Smart phones such as the Apple iPhone or Motorola Droid provide not only extremely capable hardware, but also mature OSs to develop the BCI application. Rich user interfaces could be designed to monitor neural waveforms on each channel and data could be offloaded through the high-bandwidth WiFi communication protocol or stored on the solid-state disk in the device. Both devices provide USB communication through which data could be transmitted from a small, custom board containing analog filters and ADC circuitry.

4.3 Future Work

Unlike the device presented here, many implantable BCI designs offload recorded data to a personal computer through a wireless data transmission interface such as IR (Jackson, Moritz et al. 2006) or RF (Rizk, Bossetti et al. 2009). Other members in Dr. Barbour's lab are currently designing a wireless transmission board to extract waveform shape and spike time from our device to monitor the on-board processing and control while leaving the autonomous behavior of the implant intact. Due to the extensible nature of our firmware, this integration will likely be straightforward. The ZigBee RF protocol is the current leading candidate protocol because of its low power usage and data throughput efficiency (Lee, Su et al. 2007). Communication could also allow the user to set spike detection parameters dynamically, possibly to isolate different units within a single channel of data. The normal closed-loop behavior of the implant would not be affected by this interface. Presently, this type of monitoring can be performed at the beginning of an experiment by connecting an interface cable to the header on the circuit board. Once the device is programmed it can then be disconnected and allowed to perform its task(s) autonomously while the animal moves about freely.

Because the spike detection algorithms can easily be swapped for others at run time, one can imagine building procedures into the application layer that perform intelligent pattern recognition that would choose the least complex (and therefore, quickest executing) detection algorithm from a set of implemented algorithms based upon waveform statistics. From our perspective, BCI operators should not be required to adjust detection parameters manually for every channel, especially as the

number of channels increases. Users should be able to adjust parameters if algorithms fail, but as these pattern recognition algorithms mature, the need for user intervention will likely decrease.

As the semiconductor industry continues expanding and improving microcontroller capabilities, the spike detection algorithms too expensive to run in current low-power BCI designs will eventually be implemented in small microcontrollers. This work suggests that while specific spike detection algorithms are important in BCI designs, a flexible and well-designed architecture will provide room for expansion, customization and growth that is currently not possible in BCI designs. As we explore the scientific questions and clinical applications that BCIs provide, those BCI designs most flexible will be most useful to investigators.

Appendix A Source Code

A.1 Organization

The source code for this project is divided into seven files: PNCS_App_desk.c, PNCS_App_msp.c, PNCS_App_common.h, SB_Objects.c, PNCS_common.h, PNCS_Channel.c, and PNCS_Q.c. PNCS_App_desk.c and PNCS_App_msp.c are the highest level files defining the entry point for the code when compiled for Windows and the MSP430 platform, respectively. PNCS_App_common.h and SB_Objects.c provide common global variables and application-level data structures. PNCS_common.h provides common variables to the channel and buffer data structures, which are defined in PNCS_Channel.c and PNCS_Q.c.

A.2 PNCS_App_desk.c

```
/**
@defgroup Application App-Specific
**/
#ifndef PNCS_APP_DESK_
#define PNCS_APP_DESK_
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "PNCS_App_common.h"
#include "PNCS_common.h"
#include "SB_Objects.c"

PNCS_Channel *c0;
SB_Data *d0;
FILE *fin;
FILE *fout;
void (*TAL) (PNCS_Q *, void *);
bool (*TFL) (PNCS_Q *, void *);
void *(*CUL) (void *);
void (*TAH) (PNCS_Q *, void *);
bool (*TFH) (PNCS_Q *, void *);
void *(*CUH) (void *);
void (*TAB) (PNCS_Q *, void *);
bool (*TFB) (PNCS_Q *, void *);
void *(*CUB) (void *);
void (*Off_Action)(SB_Output *);
void (*On_Action)(SB_Output *);


void Trigger_Action_Baseline (PNCS_Q *q, void *data) {
       insert_baseline(((SB_Data *)data)->b, (void *)get_data_at_ind(q,0));
}


bool Trigger_Finder_Baseline(PNCS_Q *q, void *data) {
       if ( *(Q_TYPE *)get_data_at_ind(q,0)  != 0) {
```

```
                return TRUE;
        }
        else return FALSE;
}

void * Clean_Up_Baseline (void *data) {
        if ( ((SB_Data *)data)->b->filled == TRUE) {
                process_baseline(((SB_Data *)data)->b);
                ((SB_Data *)data)->channel->trigger_action = TAL;
                ((SB_Data *)data)->channel->trigger_finder = TFL;
                ((SB_Data *)data)->channel->clean_up = CUL;
        }
        if (fout != NULL) {
                fprintf(fout, "%d\n", 0 );
        }
        return NULL;
}

/**
@author         sburns
@param          data - the user data inserted in the channel.
@remark         Copy the current q into.
@ingroup        Application
**/
void Trigger_Action_Learner (PNCS_Q *q, void *data) {
        copy_q(q, ((SB_Data *)data)->t);
}


/**
@author         sburns
@param          q - queue currently held in the channel this is called within.
@param          data - data structure that was hooked into the channel during creation.
Dereference this correctly or else...
@remark         Uses defined thresholds to determine if a waveform is of interest.
@remark         If the direction has not been set (no spikes yet), then waveform must simply
cross upper or lower threshold.
@remark         If the direction has been set, then the must waveform must cross the correct
threshold (upper threshold for upward going spikes, vice versa for downward going).
@ingroup        Application
**/
bool Trigger_Finder_Learner(PNCS_Q *q, void *data) {
        const void *d = get_data_at_ind(q, THRESHOLD_INDEX);
        if ( ((SB_Data *)data)->t->refrac == FALSE ) {
                if ( ((*(Q_TYPE *)d >= ((SB_Data *)data)->b->upper_threshold ) || (*(Q_TYPE
*)d <= ((SB_Data *)data)->b->lower_threshold )) && (*(Q_TYPE *)d != 0 ) ) {
                        ((SB_Data *)data)->t->refrac = TRUE;
                        return TRUE;
                }
                else {
                        return FALSE;
                }
        } else {
                return FALSE;
        }
}

/**
@author         sburns
@param          data - hooked data structure in PNCS_Channel.
@return         whatever data structure could be needed in main program.
@remark         This Clean_Up is used when the Channel is "learning", i.e. filling the
template. Also changes the channel into hunting mode.
@ingroup        Application
**/
void * Clean_Up_Learner (void *data) {
        if ( ((SB_Data *)data)->t->refrac == TRUE) {
                ((SB_Data *)data)->t-           >curr_refrac_count--;
```

```
		if ( ((SB_Data *)data)->t->curr_refrac_count == 0 ) {
			((SB_Data *)data)->t->curr_refrac_count = ((SB_Data *)data)->t-
>refrac_count;
			((SB_Data *)data)->t->refrac = FALSE;
		}
	}
	if ( ((SB_Data *)data)->t->filled == TRUE) {
		make_mean(((SB_Data *)data)->t);
		((SB_Data *)data)->wg->win_min = make_min_window(((SB_Data *)data)->t);
		((SB_Data *)data)->wg->win_max = make_max_window(((SB_Data *)data)->t);
		((SB_Data *)data)->channel->clean_up = CUH;
		((SB_Data *)data)->channel->trigger_action = TAH;
		((SB_Data *)data)->channel->trigger_finder = TFH;
	}
	if (fout != NULL) {
		fprintf(fout, "%d\n", 0 );
	}
	return NULL;
}

/**
@author		sburns
@param		data - the user data inserted in the channel.
@remark		Simply enabled the SB_Data
@ingroup	Application
**/
void Trigger_Action_Hunter (PNCS_Q *q,void *data) {
	if (!(((SB_Data *)data)->out->refrac_on)) {
		enable(((SB_Data *)data)->out);
	}
}

/**
@author		sburns
@param		q - queue currently held in the channel this is called within.
@param		data - data structure that was hooked into the channel during creation.
Dereference this correctly or else...
@remark		Very simple window-based method...First value must break a threshold, second
value must be greater (the waveform is increasing), and the last value must be less than a
the final window.  Window values are specific to each channel through the SB_Data.
@ingroup	Application
**/
bool Trigger_Finder_Hunter(PNCS_Q *q, void *data) {
	if (accept_wg(q, ((SB_Data *)data)->wg) ){
		return TRUE;
	} else {
		return FALSE;
	}
}


/**
@author		sburns
@param		data - hooked data structure in PNCS_Channel.
@return		whatever data structure could be needed in main program.
@remark		The hunting clean_up print 1's and 0's.
@ingroup	Application
**/
void * Clean_Up_Hunter(void *data) {
	process_output(((SB_Data *)data)->out);
	return NULL;
}


void Action_On(SB_Output *out) {
	fprintf(fout, "1\n");
}
```

```
void Action_Off(SB_Output *out) {
       fprintf(fout, "0\n");
}


/** Main function for reading spikes.txt and outputting spikes_proc.txt**/
int main(int argc, char * argv[]) {
  clock_t base_start, base_stop, hunt_start, hunt_stop, learn_start, learn_stop;
  fin = fopen("etc/spikes.txt" , "r");
        if (fin == NULL) {
        printf("Error in input file opening.");
        return 1;
  }
  fout = fopen("etc/spikes_proc.txt", "w");
  if (fout == NULL) {
        printf("Error in output file opening.");
        return 1;
  }
  Q_TYPE data = 0;
  TAB = Trigger_Action_Baseline;
  TFB = Trigger_Finder_Baseline;
  CUB = Clean_Up_Baseline;
  TAL = Trigger_Action_Learner;
        TFL = Trigger_Finder_Learner;
        CUL = Clean_Up_Learner;
  TAH = Trigger_Action_Hunter;
        TFH = Trigger_Finder_Hunter;
        CUH = Clean_Up_Hunter;
        On_Action = Action_On;
        Off_Action = Action_Off;
  d0 = new_data(1, 0, NUM_TEMPLATE, Q_SIZE, 0, BASELINE_LENGTH, On_Action, Off_Action);
        c0 = new_channel(Q_SIZE, TAB, TFB, CUB);
  if (d0 == NULL || c0 == NULL) {
        printf("Error in data allocation.");
        return 1;
  }
  sync(c0, d0);
  long int i = 0;
  long int num_samples = 1547852;
  printf("Setting baseline...\n");
  base_start = clock();
  while (d0->b->filled == FALSE) {
        fscanf(fin, "%u", &data);
        insert_and_process(c0, &data);
        i++;
  }
  base_stop = clock();
  printf("Baseline set.\n");
  print_baseline(d0->b);
  printf("Learning...\n");
  learn_start = clock();
  while ( d0->t->filled == FALSE) {
        fscanf(fin, "%u", &data);
        insert_and_process(c0, &data);
        i++;
  }
  learn_stop = clock();
  printf("Learning complete.\n");
  /**print diagnostics**/
  print_template(d0->t);
  printf("\nMax Window:");
  print_window(d0->wg->win_max);
  printf("\nMin Window:");
  print_window(d0->wg->win_min);
  /**Switch to hunting mode done in CUL**/
  printf("Hunting...\n");
  free_template(d0->t);
  hunt_start = clock();
```

```
  while (i < num_samples) {
        fscanf(fin, "%u", &data);
        insert_and_process(c0, &data);
        i++;
  }
  hunt_stop = clock();
  printf("Hunting complete.\n");
  printf("Baseline run time: %f sec \n", (double)(base_stop - base_start)/CLOCKS_PER_SEC);
  printf("Learn run time: %f sec \n", (double)(learn_stoplearn_start)/CLOCKS_PER_SEC);
  printf("Hunt run time: %f sec \n", (double)(hunt_stop - hunt_start) /CLOCKS_PER_SEC);
  return 0;
}
#endif
```

A.3 PNCS_App_msp.c

```
/**
@defgroup Application App-Specific
**/

#ifndef PNCS_APP_MSP_
#define PNCS_APP_MSP_
#include <stdio.h>
#include <stdlib.h>
#include "PNCS_App_common.h"
#include "PNCS_common.h"
#include "SB_Objects.c"
#define __MSP430F5438__                                  /** TODO: KEEP THIS define up-to-
date**/
#include <msp430.h>

/**
@author        sburns
@remark        Current operating frequency
@todo          Keep up to date.
@ingroup       Application
**/
#define CYCL_FREQ      17981000

/**
@author        sburns
@remark        Sampling frequency (Hz)
@ingroup       Application
**/
#define SAMPFREQ 8000

/**
@author        sburns
@remark        Used by the timer
@ingroup       Application
**/
#define PERIOD (CYCL_FREQ / SAMPFREQ)

PNCS_Channel *c0;
PNCS_Channel *c1;
PNCS_Channel *c2;
PNCS_Channel *c3;
SB_Data *d0;
SB_Data *d1;
SB_Data *d2;
SB_Data *d3;
bool (*TFL)(PNCS_Q *, void *);
void (*TAL)(PNCS_Q *,void *);
void *(*CUL)(void *);
bool (*TFH)(PNCS_Q *, void *);
void (*TAH)(PNCS_Q *,void *);
```

```
void *(*CUH)(void *);
void (*TAB) (PNCS_Q *, void *);
bool (*TFB) (PNCS_Q *, void *);
void *(*CUB) (void *);
void (*On_Action)(SB_Output *);
void (*Off_Action)(SB_Output *);

void Trigger_Action_Baseline(PNCS_Q *q, void *data) {
	insert_baseline(((SB_Data *)data)->b, (void *)get_data_at_ind(q,0));
}

bool Trigger_Finder_Baseline(PNCS_Q *q, void *data) {
	if ( *(Q_TYPE *)get_data_at_ind(q,0) != 0) {
		return TRUE;
	}
	else return FALSE;
}

void * Clean_Up_Baseline (void *data) {
	if ( ((SB_Data *)data)->b->filled == TRUE) {
		process_baseline(((SB_Data *)data)->b);
		((SB_Data *)data)->channel->trigger_action = TAL;
		((SB_Data *)data)->channel->trigger_finder = TFL;
		((SB_Data *)data)->channel->clean_up = CUL;
	}
	return NULL;
}

/**
@author		sburns
@param		data - the user data inserted in the channel.
@remark		Copy the current q into.
@remark		Also determines the direction of spike (only initially though).
@ingroup	Application
**/
void Trigger_Action_Learner (PNCS_Q *q, void *data) {
	copy_q(q, ((SB_Data *)data)->t);
}

/**
@author		sburns
@param		q - queue currently held in the channel this is called within.
@param		data - data structure that was hooked into the channel during creation.
Dereference this correctly or else...
@remark		Uses defined thresholds to determine if a waveform is of interest.
@remark		If the direction has not been set (no spikes yet), then waveform must simply
cross upper or lower threshold.
@remark		If the direction has been set, then the must waveform must cross the correct
threshold (upper threshold for upward going spikes, vice versa for downward going).
@ingroup	Application
**/
bool Trigger_Finder_Learner(PNCS_Q *q, void *data) {
	const void *d = get_data_at_ind(q, THRESHOLD_INDEX);
	if ( ((SB_Data *)data)->t->refrac == FALSE ) {
		if ( ((*(Q_TYPE *)d >= ((SB_Data *)data)->b->upper_threshold ) || (*(Q_TYPE
*)d <= ((SB_Data *)data)->b->lower_threshold )) && (*(Q_TYPE *)d != 0 ) ) {
			((SB_Data *)data)->t->refrac = TRUE;
			return TRUE;
		}
		else {
			return FALSE;
		}
	} else {
		return FALSE;
	}
}

/**
```

```
@author        sburns
@param         data - hooked data structure in PNCS_Channel.
@return        whatever data structure could be needed in main program.
@remark        This Clean_Up is used when the Channel is "learning", i.e. filling the
template.
@remark        Resets the refractory period when necessary.
@remark        Also changes the channel in hunting mode.
@ingroup       Application
**/
void * Clean_Up_Learner (void *data) {
       if ( ((SB_Data *)data)->t->refrac == TRUE) {
              ((SB_Data *)data)->t->curr_refrac_count--;
              if ( ((SB_Data *)data)->t->curr_refrac_count == 0 ) {
                     ((SB_Data *)data)->t->curr_refrac_count = ((SB_Data *)data)->t-
>refrac_count;
                     ((SB_Data *)data)->t->refrac = FALSE;
              }
       }
       if ( ((SB_Data *)data)->t->filled == TRUE) {
              make_mean(((SB_Data *)data)->t);
              ((SB_Data *)data)->wg->win_min = make_min_window(((SB_Data *)data)->t);
              ((SB_Data *)data)->wg->win_max = make_max_window(((SB_Data *)data)->t);
              ((SB_Data *)data)->channel->clean_up = CUH;
              ((SB_Data *)data)->channel->trigger_action = TAH;
              ((SB_Data *)data)->channel->trigger_finder = TFH;
       }
       return NULL;
}

/**
@author        sburns
@param         data - the user data inserted in the channel.
@remark        Simply enabled the SB_Data
@ingroup       Application
**/
void Trigger_Action_Hunter (PNCS_Q *q, void *data) {
       if (!(((SB_Data *)data)->out->refrac_on)) {
              enable(((SB_Data *)data)->out);
       }
}

/**
@author        sburns
@param         q - queue currently held in the channel this is called within.
@param         data - data structure that was hooked into the channel during creation.
Dereference this correctly or else...
@remark        Very simple window-based method...First value must break a threshold, second
value must be greater (the waveform is increasing), and the last value must be less than a
the final window.  Window values are specific to each channel through the SB_Data.
@ingroup       Application
**/
bool Trigger_Finder_Hunter(PNCS_Q *q, void *data) {
       if (accept_wg(q, ((SB_Data *)data)->wg) ){
                     return TRUE;
       } else {
              return FALSE;
       }
}

void Action_On(SB_Output *out) {
       P1OUT |= (out->on_bit);
}

void Action_Off(SB_Output *out) {
       P1OUT &= ~(out->on_bit);
}

/**
```

```
@author        sburns
@return        User generated output.
@param         data - is of the same type that was inserted as user_data in the channel
that is using this function. Dereference it accordingly.
@ingroup       Application
**/
void * Clean_Up_Hunter (void *data) {
       process_output(((SB_Data *)data)->out);
       return NULL;
}

/**
@author               sburns
@remark               Place all processor dependent settings in this method, is called
before any looping occurs.
@remark               When adding a channel, add it to the ADC12 channel sequence, set it
in P6SEL and P1DIR
@ingroup              Application
**/
void Initialize() {
       WDTCTL = WDTPW + WDTHOLD;                    // Stop WDT
       //CLOCK
       UCSCTL0 = 0x00;                              // Set lowest possible DCOx, MODx
       UCSCTL1 = DCORSEL_6;                         // Select range for 12MHz operation
       UCSCTL2 = 0x224;                            // Set DCO Multiplier for 16MHz (17.9
MHz)
       UCSCTL4 = SELS_3 + SELM_3;                   // Set MCLK = SMCLK = DCOCLK
       P11DIR = BIT2 + BIT1 + BIT0;                                 // P11.2,1,0 to output
direction
       P11SEL = BIT2 + BIT1 + BIT0;                                 // P11.2 to output
SMCLK, P11.1
       //ADC12 Settings
       ADC12CTL0 = ADC12SHT0_0  + ADC12ON + ADC12MSC;                           //
Sampling time, ADC12 on, Multiple sample/conv.
       ADC12CTL1 = ADC12SHP + ADC12SHS_1 + ADC12CONSEQ_1;           // Use pulse mode,
timerA0 output signal, single sequence
       ADC12MCTL0 = ADC12INCH_0;
       ADC12MCTL2 = ADC12INCH_2;
       ADC12MCTL4 = ADC12INCH_4;
       ADC12MCTL6 = ADC12INCH_6 + ADC12EOS;
       ADC12IE = BIT6;
       ADC12CTL0 |= ADC12ENC;
       //PORT DIRECTIONS
       P6SEL |= 0xFF;                                       // P6 ADC option select
       P1DIR |= 0xFF;                                               // P1 Output used by
SB_Data and Clean_Up
       //TimerA0 settings
       TA0CTL = TASSEL_2 + MC__UP;                                  //SMCLK, Up to CCR0,
       TA0CCTL0 = OUTMOD_4;                                         //Toggle mode
       TA0CCR0 = PERIOD / 2;                                        //using toggle mode,
divide by two to achieve same freq
}

int main(void) {
       P1OUT = 0;
       P2DIR |= 0xFF;
       TFL = Trigger_Finder_Learner;
       TAL = Trigger_Action_Learner;
       CUL = Clean_Up_Learner;
       TFH = Trigger_Finder_Hunter;
       TAH = Trigger_Action_Hunter;
       CUH = Clean_Up_Hunter;
       TFB = Trigger_Finder_Baseline;
       TAB = Trigger_Action_Baseline;
       CUB = Clean_Up_Baseline;
       On_Action = Action_On;
       Off_Action = Action_Off;
```

```
        d0 = new_data(1, BIT1, NUM_TEMPLATE, Q_SIZE, 0, BASELINE_LENGTH, On_Action,
Off_Action);
        d1 = new_data(1, BIT2, NUM_TEMPLATE, Q_SIZE, 0, BASELINE_LENGTH, On_Action,
Off_Action);
        d2 = new_data(1, BIT4, NUM_TEMPLATE, Q_SIZE, 0, BASELINE_LENGTH, On_Action,
Off_Action);
        d3 = new_data(1, BIT6, NUM_TEMPLATE, Q_SIZE, 0, BASELINE_LENGTH, On_Action,
Off_Action);
        c0 = new_channel(Q_SIZE,TAB,TFB,CUB);
        c1 = new_channel(Q_SIZE,TAB,TFB,CUB);
        c2 = new_channel(Q_SIZE,TAB,TFB,CUB);
        c3 = new_channel(Q_SIZE,TAB,TFB,CUB);
        sync(c0,d0);
        sync(c1,d1);
        sync(c2,d2);
        sync(c3,d3);
        P4DIR |= 0xFF;
        Initialize();
        _BIS_SR(GIE + LPM0_bits);
}

// ADC12 interrupt service routine
#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR (void)
{
        P4OUT |= BIT0;
        insert_and_process(c0, (void *)&ADC12MEM0);
        P4OUT &= ~BIT0;
        P4OUT |= BIT1;
        insert_and_process(c1, (void *)&ADC12MEM2);
        P4OUT &= ~BIT1;
        P4OUT |= BIT2;
        insert_and_process(c2, (void *)&ADC12MEM4);
        P4OUT &= ~BIT2;
        P4OUT |= BIT3;
        insert_and_process(c3, (void *)&ADC12MEM6);
        P4OUT &= ~BIT3;
    ADC12CTL0 |= ADC12ENC;
    ADC12IFG = 0;
}

#endif
```

A.4 PNCS_App_common.h

```
#ifndef PNCS_APP_COMMON_H_
#define PNCS_APP_COMMON_H_


#ifdef PNCS_APP_DESK_
/**
@author        sburns
@remark        When set, program will log information to standard output.  Clear for
release compilation.
@ingroup       Application
**/
#define DEBUG 1

/**

@author        sburns
@remark        Use this definition in new_data and new_channel.
@todo          Keep this inilne w/ SAMPFREQ (to catch around 1ms).
@ingroup       Application
**/
#define Q_SIZE 16
```

```
#endif
#ifdef PNCS_APP_MSP_
/**
@author        sburns
@remark        When set, program will log information to standard output.  Clear for
release compilation.
@ingroup       Application
**/
#define DEBUG 0

/**
@author        sburns
@remark        Use this definition in new_data and new_channel.
@todo          Keep this inilne w/ SAMPFREQ (to catch around 1ms).
@ingroup       Application
**/
#define Q_SIZE 8
#endif



/**
@author        sburns
@remark        This type must be kept in agreement with the platform it's deployed on.
@ingroup       Application
**/
#define Q_TYPE unsigned int




/**
@author        sburns
@remark        Use in new_template
@ingroup       Application
**/
#define REFRACTORY_CYCLES (Q_SIZE * 2)

/**
@author        sburns
@remark        Use this definition in new_data.
@todo          Keep update w/ Q_SIZE.
@remark        Power of 2 <= 256.
@ingroup       Application
**/
#define NUM_TEMPLATE 16

/**
@author        sburns
@remark        Use in Trigger_Finder_Learner
@ingroup
**/
#define THRESHOLD_INDEX 2


/**
@author        sburns
@param
@return
@remark
@ingroup
**/
#define BASELINE_LENGTH 32

#endif /*PNCS_APP_COMMON_H_*/
```

A.5 SB_Objects.c

```
/**
@defgroup Application App-Specific
**/

#ifndef __SB_OBJECTS__
#define __SB_OBJECTS__
#include "PNCS_common.h"

/**
@author        sburns
@remark        Simple struct to abstract a window for use in Trigger_Finder
@ingroup       Application
**/
typedef struct {
       Q_TYPE index;                /*!< Index >*/
       Q_TYPE value;                /*!< Window Value >*/
       bool above;                         /*!< Direction >*/
} SB_Window;

/**
@author        sburns
@param         ind - index that window is temporally placed at within PNCS_Q.
@param         val - value representing voltage that signal is compared against
@param         dir - type of window (down going or up going)
@remark        Used to create SB_Window.  Caller must check for null pointer.
@memberof<SB_Window>
**/
SB_Window * new_window(Q_TYPE ind, Q_TYPE val, bool dir) {
       SB_Window *window = malloc(sizeof(SB_Window));
       if (!window){
              return NULL;
       }
       window->index = ind;
       window->value = val;
       window->above = dir;
       return window;
}

/**
@author        sburns
@param         q - PNCS_Q which holds data to be checked against the window.
@param         w - SB_Window that checks the PNCS_Q.
@remark        Simple logic.
@memberof<SB_Window>
**/
bool accept(PNCS_Q *q, SB_Window *w) {
       Q_TYPE *val = (Q_TYPE *) get_data_at_ind(q,w->index);
       if (((w->above == TRUE) && (*val >= w->value)) || ((w->above == FALSE) && (*val <=
w->value))) {
              return TRUE;
       }
       else return FALSE;
}

/**
@author        sburns
@param         w - window to be printed to stdout.
@remark        Used for debugging.
@ingroup       Application.
@memberof<SB_Window>
**/
void print_window(SB_Window *w) {
       if (DEBUG) {
              printf("Index:\t%u\tValue:\t%u\tDirection:\t%u\n", w->index,w->value, w-
```

```
>above);
        }
}

/**
@author         sburns
@remark         Structure to hold both windows.  Pass the structure to an accept function.
@ingroup        Application
**/
typedef struct {
        SB_Window *win_min;
        SB_Window *win_max;
} SB_Window_Group;

/**
@author         sburns
@return         Valid WG pointer or null
@ingroup        Application
@memberof<SB_Window_Group>
**/
SB_Window_Group * new_wg() {
        SB_Window_Group *wg = malloc(sizeof(SB_Window_Group));
        wg->win_max = NULL;
        wg->win_min = NULL;
        return wg;
}


/**
@author         sburns
@param          q - PNCS_Q currently
@param          WG - SB_Window_Group associated with channel
@return         bool - t if both windows fit q, f otherwise
@remark         Simplifies writing TFH.
@ingroup        Application
@memberof<SB_Window_Group>
**/
bool accept_wg(PNCS_Q *q, SB_Window_Group *WG) {
        if (accept(q, WG->win_max) && accept(q, WG->win_min)) {
                return TRUE;
        }
        else return FALSE;
}



/**
@author         sburns
@remark         Structure for storing an N (row) x M (column) matrix. When matrix is filled,
a mean is computed, which is used to create optimal windows.
@ingroup        Application
**/
typedef struct {
        Q_TYPE *data;                                           /*!< Pointer to hold NxM matrix
(linear)>*/
        unsigned long *mean;                            /*!< Pointer to hold final mean >*/
        Q_TYPE *curr;                                           /*!< Pointer to next data to be
filled >*/
        Q_TYPE *end;                                            /*!< Pointer to end of array,
used for filled >*/
                                                                        /*!< number of rows >*/
        unsigned int M;                                                 /*!< number of columns
>*/
        bool filled;                                            /*!< filled? >*/
        unsigned int N_pow2;                            /*!< 2^N_pow2 = N, used in make mean
for fast division >*/
        bool refrac;                                            /*!< Has a threshold recently
been broken? >*/
```

```
	unsigned int refrac_count;			/*!< Number of cycles before a spike
will be reported after an initial >*/
	unsigned int curr_refrac_count;		/*!< Current count, if == to count,
then current_spike == FALSE >*/
} SB_Template;

unsigned int find_pow2(unsigned int N) {
	unsigned int ans;
	switch (N) {
		case 1: {
			ans = 0;
			break;
		}
		case 2: {
			ans = 1;
			break;
		}
		case 4: {
			ans = 2;
			break;
		}
		case 8: {
			ans = 3;
			break;
		}
		case 16: {
			ans = 4;
			break;
		}
		case 32: {
			ans = 5;
			break;
		}
		case 64: {
			ans = 6;
			break;
		}
		case 128: {
			ans = 7;
			break;
		}
		case 256: {
			ans = 8;
			break;
		}
	}
	return ans;
}


/**
@author		sburns
@param		N - number of rows
@param		M - number of columns
@memberof<SB_Template>
**/
SB_Template * new_template(Q_TYPE N, Q_TYPE M) {
	int i;
	SB_Template *temp = malloc(sizeof(SB_Template));
	if (!temp) {
		return NULL;
	}
	temp->N = N;
	temp->M = M;
	temp->data = malloc(N * M * sizeof(Q_TYPE));
	if (!temp->data){
		return NULL;
	}
```

```
	temp->mean = malloc(M * sizeof(long));
	if (!temp->mean) {
		return NULL;
	}
	temp->curr = temp->data;
	temp->filled = FALSE;
	for (i = 0; i < temp->M; i++) {
		temp->mean[i] = 0;
	}
	temp->end = &(temp->data[temp->N * temp->M - 1]);
	temp->N_pow2 = find_pow2(N);
	temp->refrac_count = REFRACTORY_CYCLES;
	temp->curr_refrac_count = temp->refrac_count;
	temp->refrac = FALSE;

	return temp;
}

/**
@author		sburns
@param		q - queue to be copied into template matrix
@param		t - template to hold q
@remark		Copy a PNCS_Q (one of interest) into the next row in matrix.
@memberof<SB_Template>
**/
status copy_q(PNCS_Q *q, SB_Template *t) {
	int i = 0;

	if (  t->filled == TRUE) {
		if (DEBUG) {
			printf("Error in copy_q.\n");
		}
		return ERROR;
	}
	for (; i < t->M; i++) {
		*(t->curr) = *(Q_TYPE *)get_data_at_ind(q, i);
		t->curr++;
	}
	if ( t->curr >= t->end) {
		t->filled = TRUE;
	}

	return OK;
}

/**
@author		sburns
@param		t - SB_Template that has been filled.
@remark		When full, the template means should be calculated to make the average
waveform.
@memberof<SB_Template>
**/
status make_mean(SB_Template *t) {
	int ii, jj;
	if ( t->filled == FALSE) {
		if (DEBUG) {
			printf("Error in make_mean.\n");
		}
		return ERROR;
	}
	t->curr = t->data;
	for (ii = 0; ii < t->N; ii++) {
		for (jj = 0; jj < t->M; jj++) {
			t->mean[jj] += *(t->curr);
			t->curr++;
		}
	}
	for (ii = 0; ii < t->M; ii++) {
```

```
		t->mean[ii] = t->mean[ii] >> t->N_pow2;
	}
	return OK;
}

/**
@author		sburns
@param		t - template to build max window off.
@return		an SB_Window that will look for the max peak of the waveform.
@memberof<SB_Template>
**/
SB_Window * make_max_window(SB_Template *t) {
	unsigned int ind, max, i = 0;
	ind = 0;
	max = t->mean[0];
	for (i = 1; i < t->M; i++) {
		if (t->mean[i] > max) {
			ind = i;
			max = t->mean[i];
		}
	}
	return new_window(ind, max-75, TRUE);
}

/**
@author		sburns
@param		t - template to build max window off.
@return		an SB_Window that will look for the min peak of the waveform.
@memberof<SB_Template>
**/
SB_Window * make_min_window(SB_Template *t) {
	unsigned int ind, min, i = 0;
	ind = 0;
	min = t->mean[0];
	for (i = 1; i < t->M; i++) {
		if (t->mean[i] < min) {
			ind = i;
			min = t->mean[i];
		}
	}
	return new_window(ind, min+75, FALSE);
}

/**
@author		sburns
@param		t - template to be printed to stdout
@remark		Used for debugging purposes.  Should only be called when template is full.
@ingroup	Application
@memberof<SB_Template>
**/
void print_template(SB_Template *t) {
	if (DEBUG) {
		Q_TYPE *curr = t->data;
		unsigned long *pmean = t->mean;
		int ii, jj;
		if (t->filled == FALSE ) {
			printf("Error in print_template, template is not full.");
			return;
		}
		printf("Full Templates:\n");
		for (ii = 0; ii < t->N; ii++) {
			for (jj = 0; jj < t->M; jj++) {
				printf("%u\t", *(curr));
				curr++;
			}
			printf("\n");
		}
		printf("Mean template:\n");
```

```
                for (ii = 0; ii < t->M; ii++) {
                        printf("%ld\t", *(pmean));
                        pmean++;
                }
        }
}

/**
@author         sburns
@param          t - template to be freed
@remark         free the template
@ingroup        Application
@memberof<SB_Template>
**/
void free_template(SB_Template *t) {
        free(t->data);
        free(t->mean);
        free(t);
}

typedef struct SB_Output SB_Output;

/**
@author         sburns
@remark         Structure for grouping output parameters together
@ingroup        Application
**/
struct SB_Output{
        bool enabled;                                   /*!< simple boolean.*/
        bool is_on;                                             /*!< set after delay and
cleared after reset. >*/
        bool delay_on;
        bool refrac_on;
        char on_bit;                                    /*!< Currently, the pin on PORT1 that
will be turned on*/
        int cycles;                                             /*!< constant value for how
many cycles an action will be performed for.*/
        int current_cycles;                             /*!< if enabled is true, action will
be taken for this many more cycles*/
        int delay;                                              /*!< Delay between trigger and
output >*/
        int current_delay;                              /*!< current delay >*/
        int refrac;
        int current_refrac;
        void (*on_action)(SB_Output *);
        void (*off_action)(SB_Output *);
} ;


/**
@author         sburns
@param          cycles - how many loops output pulse will be on.
@param          bit - P2OUT pin
@param          delay - amount of cycles before pulse is turned on after trigger.
@return         Valid SB_Output structure or null
@ingroup        Application
@memberof<SB_Output>
**/
SB_Output * new_out(int cycles, char bit, int delay, void (*on_action)(SB_Output *),void
(*off_action)(SB_Output *)) {
        SB_Output *out = malloc(sizeof(SB_Output));
        if (out == NULL) {
                return NULL;
        }
        out->enabled = FALSE;
        out->current_cycles = cycles;
        out->cycles = cycles;
        out->delay = delay;
```

```
        out->current_delay = delay;
        out->on_bit = bit;
        out->is_on = FALSE;
        out->refrac = Q_SIZE;
        out->current_refrac = Q_SIZE;
        out->refrac_on = FALSE;
        out->on_action = on_action;
        out->off_action = off_action;
        return out;
}


/**
@author         sburns
@param
@return
@remark
@ingroup
**/
void enable(SB_Output *out){
        out->enabled = TRUE;
        out->delay_on = TRUE;
}

/**
@author         sburns
@param
@return
@remark         Handle delay, then cycles, then refrac
@ingroup
**/
void process_output(SB_Output *out) {
        if (out->enabled == TRUE) {
                if (out->delay_on == TRUE) {
                        out->off_action(out);
                        out->current_delay--;
                        if (out->current_delay < 0) {
                                out->delay_on = FALSE;
                                out->current_delay = out->delay;
                                out->is_on = TRUE;
                        }
                }
                else {
                        if (out->is_on == TRUE ) {
                                out->on_action(out);
                                out->current_cycles--;
                                if (out->current_cycles < 0) {
                                        out->is_on = FALSE;
                                        out->current_cycles = out->cycles;
                                        out->refrac_on = TRUE;
                                }
                        }
                        if (out->refrac_on == TRUE) {
                                out->off_action(out);
                                out->current_refrac--;
                                if (out->current_refrac < 0) {
                                        out->enabled = FALSE;
                                        out->refrac_on = FALSE;
                                        out->current_refrac = out->refrac;
                                }
                        }
                }
        }
        else {
                out->off_action(out);
        }

}
```

```
/**
@author        sburns
@remark        Struct for determining level of noise at the onset of running the program.
@remark        Sets upper and lower thresholds for spike detection after computing
statistics.
@ingroup       Application
**/
typedef struct {
        Q_TYPE *data;
        Q_TYPE *curr;
        Q_TYPE *last;
        unsigned int size;
        unsigned int size_pow2;
        bool filled;
        unsigned long mean;
        unsigned int sd;
        unsigned int upper_threshold;
        unsigned int lower_threshold;
} SB_Baseline;


SB_Baseline * new_baseline(unsigned int size) {
        SB_Baseline *base = malloc(sizeof(SB_Baseline));
        if (base == NULL) {
                return base;
        }
        base->data = malloc(size * sizeof(Q_TYPE));
        if (base->data == NULL) {
                free(base);
                return NULL;
        }
        base->curr = base->data;
        base->last = &(base->data[size - 1]);
        base->size = size;
        base->size_pow2 = find_pow2(size);
        base->filled = FALSE;
        base->lower_threshold = 0;
        base->upper_threshold = 0;
        base->sd = 0;
        base->mean = 0;
        return base;
}


/**
@author        sburns
@param         b - SB_Baseline struct
@param         data - void * that is of type of data throught out all structures
@remark        Simple insertion, flips bool when array is filled.
@ingroup       Application
@memberof<SB_Baseline>
**/
void insert_baseline(SB_Baseline *b, void *data) {
        *(b->curr) = * (Q_TYPE *)data;
        b->curr++;
        if (b->curr > b->last) {
                b->filled = TRUE;
        }
}


/**
@author        sburns
@param         b - SB_Baseline that has already been filled
@return        OK if all proceeds as planned, ERROR otherwise
```

```
@remark        Approximates standard deviation by knowing Vpp of noise is ~4 times standard
deviation.
@remark        creates thresholds by +- 8*std + mean
@ingroup       Application
@memberof<SB_Baseline>
**/
status process_baseline(SB_Baseline *b) {
       unsigned int i, max = 0, min = 0;
       if (b->filled != TRUE) {
              return ERROR;
       }
       b->curr = b->data;
       for (i = 0; i < b->size; i++) {
              b->mean += *(b->curr);
              if (*(b->curr) > max) {
                     max = *(b->curr);
              }
              if (*(b->curr) < min) {
                     min = *(b->curr);
              }
              b->curr++;
       }
       b->mean = b->mean >> b->size_pow2;
       b->sd = (max - min) >> 3;
       b->upper_threshold = b->mean + (b->sd << 1);
       b->lower_threshold = b->mean - (b->sd << 1);
       return OK;
}

/**
@author        sburns
@param         b - baseline to be printed to stdout
@remark        simple debugging function
@ingroup       Application
@memberof<SB_Baseline>
**/
void print_baseline(SB_Baseline *b){
       if (DEBUG) {
              printf("Baseline:\n");
              printf("Mean: %u\tSD:%u\t\n", b->mean, b->sd);
              printf("Upper: %u\tLower: %u\t\n", b->upper_threshold, b->lower_threshold);

       }
}


/**
@author        sburns
@remark        Simple struct to keep track of an action over cycles. Provides customizable
window values for each channel.
@ingroup       Application
**/
typedef struct {
       SB_Output *out;                              /*!< Struct coordinating output params
>*/
       SB_Window_Group *wg;          /*!< Struct for holding both windows >*/
       SB_Template *t;                              /*!< template to build windows>*/
       SB_Baseline *b;                              /*!< baseline structure >*/
       PNCS_Channel *channel;        /*!< reference to encompassing structure>*/
} SB_Data;



/**
@author        sburns
@param         cycles - how many cycles the action will take place for once started. Should
be greater than 0, otherwise nothing would happen.
@remark        If using this for pulses with   sampling period resolution, pulse width will
```

```
be (cycles - 1) samples. May want to correct this in the future, but for now, it works.
@param         bit - pin number on PORT1 that is tied to this data structure.
@param         threshold - In this application, the threshold above which the waveform must
pass initially.
@param         win1 - The first window value.  Data must be greater than this.
@param         win2 - The second window value.  Data must be less than this.
@return        Initialized SB_Data or null pointer.
@ingroup       Application
@memberof<SB_Data>
**/
SB_Data * new_data(int cycles, char bit, int N, int M, int delay, unsigned int base_size,
void (*on_action)(SB_Output *), void (*off_action)(SB_Output *)) {
        SB_Data *SB = malloc(sizeof(SB_Data));
        SB_Template *temp = new_template(N, M);
        SB_Window_Group *wg = new_wg();
        SB_Output *out = new_out(cycles, bit, delay, on_action, off_action);
        SB_Baseline *b = new_baseline( base_size);
        if (SB == NULL || temp == NULL || wg == NULL || out == NULL) {
                if (DEBUG) {
                        printf("Failure to allocate an SB_Data.");
                        return NULL;
                }
        }
        SB->t = temp;
        SB->wg = wg;
        SB->out = out;
        SB->b = b;
        return SB;
}

/**
@author        sburns
@param         SB - SB_Data to be deleted.
@remark        Simple free.
@ingroup       Application
@memberof<SB_Data>
**/
void free_SB(SB_Data *SB){
        free(SB);
}


/**
@author        sburns
@param         c - channel to be attached to data
@param         d - data to be attached to channel
@remark        Before allocation, global pointers are null, so this synchronization should
be done after allocating both channel and data.
@ingroup       Application
**/
void sync(PNCS_Channel *c, SB_Data *d) {
        c->user_data = d;
        d->channel = c;
}


#endif
```

A.6 PNCS_common.h

```
/**
@defgroup PNCS PNCS
**/


/**
```

```
@ingroup PNCS
**/
#ifndef PNCS_COMMON_H_
#define PNCS_COMMON_H_

typedef enum  er status;
typedef enum {FALSE = 0, TRUE = 1} bool;
typedef struct PNCS_Channel PNCS_Channel;
#include "PNCS_Channel.c"

extern void Trigger_Action(PNCS_Q *q, void *data);
extern bool Trigger_Finder(PNCS_Q *q, void *data);
extern void * Clean_Up(void *data);
#endif
```

A.7 PNCS_Channel.c

```
/**
@ingroup PNCS
**/

#ifndef PNCS_CHANNEL_C_
#define PNCS_CHANNEL_C_

#include <stdlib.h>
#include <stdio.h>

#include "PNCS_Q.c"

/**
@author        sburns
@remark        Main data structure of PNCS.
**/
struct PNCS_Channel{
       PNCS_Q *buffer;
       /*!< data queue*/
       void (*trigger_action)(PNCS_Q * q, void *data);            /*!< callback function
called after ratio amount of triggers have been found.*/
       bool (*trigger_finder)(PNCS_Q *q, void *data);             /*!< function used to
search for a trigger in the buffer.*/
       void *(*clean_up)(void *data);                                          /*!<
final function called in insert_and_process().*/
       void * user_data;
       /*!< user-defined data that should be held by a single channel and used in callback
functions */
} ;


/**
@author        sburns
@param         buffer_size - amount of data stored in the queue.
@param         TA - a function pointer to an action the channel will perform when given
enough triggers.
@param         TF - a function pointer the channel uses to determine if a trigger is
present.
@param         CU - a clean up function always called at the end of every
insert_and_process().
@param         data - pointer containing data that the user wants to bind to a channel. Can
be accessed in callback functions.
@return        memory containing a valid instance of PNCS_Channel or a NULL pointer.
@remark        Up to client to check for null returned pointer.
@memberof <PNCS_Channel>
**/
```

```
 PNCS_Channel * new_channel(int buffer_size, void (*TA) (PNCS_Q *q,void *data), bool (*TF)
(PNCS_Q *q, void *data), void *(*CU) (void *data)) {
        PNCS_Channel *C = malloc(sizeof( PNCS_Channel));
        if (!C) {
                if (DEBUG) {
                        printf("Could not allocate memory for PNCS_Channel.");
                }
                return NULL;
        }
        C->buffer = new_Q(buffer_size);
        if (!C->buffer) {
                if (DEBUG){
                        printf("Could not allocate PNCS_Q within PNCS_Channel.");
                }
                return NULL;
        }
        C->trigger_finder = TF;
        C->trigger_action = TA;
        C->clean_up = CU;
        return C;
}

/**
@author         sburns
@param          C - channel to be freed.
@remark         Frees the data queue, and the channel structure itself.
@memberof<PNCS_Channel>
**/
void free_channel(PNCS_Channel *C){
        free_Q(C->buffer);
        free(C);
}

/**
@author         sburns
@param          C - channel to insert data into and then process
@param          data - valid pointer of data (must be type held in the q)
@return         returns the value of C->clean_up
@remark         First inserts the data, then uses the trigger finder function to search the
buffer, if that returns true trigger_action is called.  clean_up is called no matter what.
@memberof<PNCS_Channel>
**/
void * insert_and_process(PNCS_Channel *C, void *data){
        insert(C->buffer, data);
        if (C->trigger_finder(C->buffer, C->user_data))  {
                C->trigger_action(C->buffer, C->user_data);
        }
        return C->clean_up(C->user_data);
}

/**
@author         sburns
@param          C - channel to view
@remark         Used for debugging purposes.  Compiles to a stub if DEBUG is set to zero.
@memberof<PNCS_Channel>
**/
void view_channel(PNCS_Channel *C){
        if (DEBUG) {
                printf("Buffer:\n");
                view_Q(C->buffer);
                printf("\n");
                C->trigger_finder(C->buffer, C->user_data);
                C->trigger_action(C->buffer, NULL);
                C->clean_up(C->user_data);
        }
}
```

```
#endif
```

A.8 PNCS_Q.c

```
/**
@ingroup PNCS
**/

#ifndef PNCS_Q_C_
#define PNCS_Q_C_

#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>

/**
@author        sburns
@remark        Buffer structure that uses a dynamic array for minimal memory usage
**/
typedef struct {
        Q_TYPE Q_size;                      /*!< size of queue*/
        Q_TYPE *first;                      /*!< "first to be deleted", i.e. oldest data
in buffer*/
        Q_TYPE *last;                       /*!< last data point in array (not necessarily
last data temporally) >*/
        Q_TYPE *data;                       /*!< data array >*/

} PNCS_Q;



/**
@author        sburns
@param         q - queue to request data from.
@param         ind - index of requested data in standard C usage (0 for first, size-1 for
last).
@return        pointer to data.
@remark        client should cast this appropriately to dereference.
@memberof<PNCS_Q>
@public
**/
const void * get_data_at_ind(PNCS_Q *q, int ind) {
        if ( (q->first + ind) <= (q->last) ) {
                return (q->first + ind);
        }
        else return (q->first - (q->Q_size - ind));
}


/**
@author        sburns
@param         q - q to be printed
@return        the current state of the q is printed to standard output.
@remark        Print the queue on standard output.  Only used for debugging purposes.
Compiles to stub when DEBUG is set to FALSE.
@memberof<PNCS_Q>
@public
**/
void view_Q(PNCS_Q *q) {
```

```
	if ( DEBUG ) {
		int i = 0;
		for (;i < q->Q_size;i++) {
			printf("%d\t", *(Q_TYPE *) get_data_at_ind(q, i));
		}
		printf("\n");
	}
}


/**
@author		sburns
@param		q - queue to insert new data into
@param		data - data to be inserted, inserted data type MUST be the defined Q_TYPE in
PNCS_App.c
@return		OK if insertion occurs normally, ERROR otherwise.
@remark		Trusts the user that *data is the same type that the queue was initially set
up to use.
@memberof<PNCS_Q>
@public
**/

void insert(PNCS_Q *q, void *data) {
	*(q->first) = *(Q_TYPE *)data;
	(q->first)++;
	if ( q->first > q->last ) {
		q->first = q->data;
	}
}


/**
@author		sburns
@param		size - amount of data points to be held in the queue at all times.
@return		a pointer to memory containing an initialized q or null pointer.
@remark		Calling function must check for null pointer. Returns a null pointer if 1)
allocation for whole structure fails, or 2) allocation for data array fails.
@memberof<PNCS_Q>
@public
**/
PNCS_Q * new_Q ( int size ){
	PNCS_Q *q = malloc(sizeof(PNCS_Q));
	if (q == NULL) {
		return q;
	}
	q->Q_size = size;
	q->data = NULL;
	q->data = malloc(size * sizeof(Q_TYPE));
	if (q->data) {
		q->first = q->data;
	}
	else {
		q = NULL;
	}
	q->last = &(q->data[q->Q_size - 1]);
	return q;
}


/**
@author			sburns
@param			q - PNCS_Q to be freed.
@return			none
@remark			Frees all the data locations in the array, frees the array, and then
frees the entire structure.
@memberof<PNCS_Q>
@public
**/
```

```
void free_Q(PNCS_Q *q) {
        free(q->data);
        q->first = NULL;
        free(q);
        q = NULL;
}


/**
@author         sburns
@param          q - the q who's last data point is desired
@return         pointer to the data (which can only be read)
@remark         returns a pointer to the last data point that can only be read.the data type
is the type of the q.
@memberof<PNCS_Q>
@public
**/
const void * get_last(PNCS_Q *q) {
        return get_data_at_ind(q,q->Q_size-1);
}

#endif /*PNCS_Q_C_*/
```

References

An, S. K., S.-I. Park, et al. (2007). "Design for a simplified cochlear implant system." IEEE Transactions on Biomedical Engineering **54**(6): 973-982.

Bewernick, B., R. Hurlemann, et al. (2009). "Nucleus Accumbens Deep Brain Stimulation Decreases Ratings of Depression and Anxiety in Treatment-Resistant Depression." Biological Psychiatry.

Bossetti, C. A., J. M. Carmena, et al. (2004). "Transmission Latencies in a Telemetry-Linked Brain-Machine Interface." IEEE Transactions on Biomedical Engineering **51**(6): 919-924.

Brummer, S. B. and M. J. Turner (1975). "Electrical stimulation of the nervous system: The principle of safe charge injection with noble metal electrodes." Bioelectrochemistry and Bioenergetics **2**(1): 13-25.

Burns, S. and D. Barbour (2009). A fully-customizable software platform for low-power brain-computer interfaces. Society for Neuroscience, Chicago, IL, 181.17.

Butterworth, S. (1930). "Theory of filter amplifiers." Experimental Wireless & the Wireless Engineer: 536-541.

Chestek, C. A., V. Gilja, et al. (2009). "HermesC: low-power wireless neural recording system in freely moving primates." IEEE Transactions on Neural Systems and Rehabilitation Engineering **17**(4): 330-338.

Doran, M. (2003). "From Research and Development at Intel: Beyond the BIOS." Intel Developer Conference.

Edwards, S., L. Luciano, et al. (1997). "Design of Embedded Systems: Formal Models, Validation, and Synthesis." Proceedings of the IEEE **85**(3): 366-390.

Georgopoulos, A., J. Lurito, et al. (1989). "Mental Rotation of the Neuronal Population Vector." Science **243**(4888): 234-236.

Grohrock, P., U. Hausler, et al. (1997). "Dual-channel telemetry system for recording vocalization-correlated neuronal activity in freely moving squirrel monkeys." Journal of Neuroscience Methods **76**: 7-13.

Hampson, R., V. Collins, et al. (2009). "A wireless recording system that utilizes Bluetooth technology to transmit neural activity in freely moving animals." Journal of Neuroscience Methods **182**(2): 195-204.

Harrison, R. R., R. J. Kier, et al. (2009). "Wireless Neural Recording With Single Low-Power Integrated Circuit." IEEE Transactions on Neural Systems and Rehabilitation Engineering **17**(4): 322-329.

Hochberg, L., M. Serruya, et al. (2006). "Neuronal ensemble control of prosthetic devices by a human with tetraplegia." Nature **442**: 164-171.

Hodgkin, A. L. and A. F. Huxley (1945). "Resting and action potentials in single nerve fibres " The Journal of Physiology **104**(2): 176-195.

Huang, D., P. Lin, et al. (2009). "Decoding human motor activity from EEG single trials for a discrete two-dimensional cursor control." Journal of Neural Engineering **6**(4).

Jackson, A., J. Mavoori, et al. (2006). "Long-term motor cortex plasticity induced by an electronic neural implant." Nature **444**: 56-60.

Jackson, A., C. T. Moritz, et al. (2006). "The Neurochip BCI: Towards a Neural Prosthesis for Upper Limb Function." IEEE Transactions on Neural Systems and Rehabilitation Engineering **14**(2): 187-190.

Joyner, R. W., M. Westerfield, et al. (1980). "Effects of cellular geometry on current flow during a propagated action potential." Biophysical Journal **31**(2): 183-194.

Karki, J. (2002). "Analysis of the Sallen-Key Architecture." Texas Instruments Application Report.

Kennedy, P., R. Bakay, et al. (2000). "Direct control of a computer from the human central nervous system." IEEE Transactions on Rehabilitation Engineering **8**(2): 198-202.

Lee, J.-S., Y.-W. Su, et al. (2007). A comparitive study of wireless protocols: Bluetooth, UWB, ZigBee, Wi-Fi. IEEE Industrial Electronics Society, Taipei, Taiwan.

Lei, Y., N. Sun, et al. (2004). "Telemetric recordings of single neuron activity and visual scenes in monkeys walking in an open field." Journal of Neuroscience Methods **30**(1-2): 35-41.

Lewicki, M. (1999). "A review of methods for spike sorting: the detection and classification of neural action potentials." Network **9**(4): 53-78.

Mavoori, J., A. Jackson, et al. (2005). "An autonomous implantable computer for neural recording and stimulation in unrestrained primates." Journal of Neuroscience Methods **148**: 71-77.

Mojarradi, M., D. Binkely, et al. (2003). "A miniaturized neuroprosthesis suitable for implantation into the brain." IEEE Transactions on Neural Systems and Rehabilitation Engineering **11**(1): 38-42.

Moore, G. E. (1965). "Cramming more components onto integrated circuits." Electronics **38**(8).

Obeso, J., C. Olanow, et al. (2001). "Deep-brain stimulation of the subthalamic nucleus or the pars interna of the globus pallidus in Parkinson's Disease." New England Journal of Medicine **345**(13): 956-963.

Pattle, R. (1971). "The external action potential of a nerve or muscle fibre in an extended medium." Phys. Med. Biol. **16**(4): 673-685.

Pease, R. A. (2008). A Comprehensive Study of the Howland Current Pump. National Semiconductor Application Notes.

Raghunathan, S., S. Gupta, et al. (2009). "The design and hardware implementation of a low-power real-time seizure detection algorithm." Journal of Neural Engineering **6**(5).

Rizk, M., C. A. Bossetti, et al. (2009). "A fully implantable 96-channel neural data acquisition system." Journal of Neural Engineering **6**.

Rolston, J. D., R. E. Gross, et al. (2009). "A low-cost multielectrode system for data acquisition enabling real-time closed-loop processing with rapid recovery from stimulation artifacts." Frontiers In Neuroscience **2**.

Salinas, E. and T. J. Sejnowski (2001). "Correlated neuronal activity and the flow of neural information." Nature Reviews Neuroscience **2**(8): 539-550.

Santhanam, G., S. Ryu, et al. (2006). "A high-performance brain-computer interface." Nature **442**(7099): 195-8.

Schmidt, E., J. McIntosh, et al. (1978). "Fine control of operantly conditioned firing patterns of cortical neurons." Experimental Neurology **61**(2): 349-369.

Smith, S. W. (1998). The Scientist's and Engineer's Guide to Digital Signal Processing. San Diego, CA, California Technical Publishing. **2009**.

Stoney, S. J., W. Thompson, et al. (1968). "Excitation of pyramidal tract cells by intracortical microstimulation: effective extent of stimulating current." J Neurophysiology **31**(5): 659-69.

Taylor, D., S. Helms Tillery, et al. (2002). "Direct cortical control of 3D neuroprosthetic devices." Science **296**(5574): 1829-1832.

Texas Instruments (2009). MSP430F543x Data Sheet. Dallas, Texas Instruments. **2009**.

Utsugi, K., A. Obata, et al. (2007). Development of an Optical Brain-machine Interface. Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE.

Vidal, J. (1973). "Toward Direct Brain-Computer Communication." Annual Review of Biophysics and Bioengineering **2**: 157-180.

Wilson, B. S., C. C. Finley, et al. (1991). "Better speech recognition with cochlear implants." Nature **352**: 236-238.

Wilson, J. and J. C. Williams (2009). "Massively parallel signal processing using the graphics processing unit for real-time brain-computer interface feature extraction." Frontiers In Neuroscience **2**.

Wolpaw, J., D. McFarland, et al. (1991). "An EEG-based brain-computer interface for cursor control." Electroencephalographic Clinical Neurophysiology **78**(3): 252-9.

Xu, S., S. K. Talwar, et al. (2004). "A multi-channel telemetry system for brain microstimulation in freely roaming animals." Journal of Neuroscience Methods **133**: 57-63.

Ye, X., P. Wang, et al. (2008). "A portable telemetry system for brain stimulation and neuronal activity recording in freely behaving small animals." Journal of Neuroscience Methods **174**: 186-193.

Yun, X., D. Kim, et al. (2007). Low-power high-resolution 32-channel neural recording system. Conference of IEEE EMBS, Lyon, France.

Vita

Scott Sparkman Burns

Date of Birth February 22, 1986

Place of Birth Montgomery, AL

Degrees B.S. Biomedical Engineering, May 2008
M.S. Biomedical Engineering, May 2010

Publications Burns SS, Barbour DL. A fully customizable software platform for low power brain-computer interfaces. Program No.181.17. 2009 Neuroscience Meeting Planner. Chicago, IL: Society for Neuroscience, 2009. Online.

Hentall ID, Burns, SS. Restorative effects of stimulating medullary raphe after spinal cord injury.J Rehabil Res Dev. 2009;46(1):109-22.

Hentall ID, Burns SS, Rodriguez ML. Intermittent electrical stimulation of the serotonergic medullary raphe by a small implanted stimulator as a means to improve recovery in rats after incomplete spinal contusion injury. Journal of Neurotrauma. 2007, 24(7): 1229-1288. doi:10.1089/neu.
2007.9972.

May 2010