Telemetric Brain Electrode Array: Wireless and Analog Subsystems

by

Johann Burgert

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

At the Massachusetts Institute of Technology

May 10, 2002

© 2002 Massachusetts Institute of Technology. All rights reserved.
Telemetric Brain Electrode Array: Wireless and Analog Subsystems

by

Johann David Burgert

Submitted to the Department of Electrical Engineering and Computer Science

May 10, 2002

in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Telemetric Electrode Array System (TEAS) is a surgically implantable, wireless device for studying neural activity in the brain and for developing brain-machine interfaces. An array of electrodes collects intra-cortical neural signals for the electronics system to process and wirelessly transfer to an external recording system. The waveforms can be analyzed in real-time and used to control a prosthesis or be stored and used in neurology research. We have designed an analog front end that amplifies the microvolt level signals before they are digitized with 12 bits of resolution at 31 kHz per channel. A digital subsection performs peak detection on the sixty-four channels to compress the data before sending them over a Bluetooth-based radio link at 725 kbit per second. The individual sections of the system have been designed and tested, but a fully integrated and implantable version has not been implemented due to time constraints.

Thesis Supervisor: Ian W. Hunter
Title: Professor of Mechanical Engineering and BioEngineering
Acknowledgements

I would like to thank the other members of the TEAS team:

- Sylvain Martel and Professor Ian Hunter for their leadership and help.
- Jan Malášek who designed the digital section, the flex connector, the ADC control system and helped with everything else.
- Colette Wiseman, Robert Dyer, Tim Fofonoff, and Marie-Maude de Denus Baillargeon who designed and built the array.
- Martin Labrecque and Thor Bjarnason who worked on speeding up the Bluetooth link.
- Fedor Danilenko who implemented the wireless recharging system prototype.
- Nicho Hatsopoulos, John Donoghue, Misha Serruya, and the rest of our collaborators at Brown who initiated and funded this project as well as gave us specifications, answered neurology questions, and helped with implementation and testing.

Thanks to John Ankcom and the rest of the ORnet team from the MIT Laboratory for Computer Science for their help in obtaining Bluetooth modules and making them work.

I would also like to thank all the members of the MIT BioInstrumentation lab for their support and the education in non EECS material.
# Table of Contents

**ABSTRACT** ................................................................................................................................. 3

**LIST OF TABLES** ............................................................................................................................. 9

**LIST OF FIGURES** ........................................................................................................................... 9

**INTRODUCTION** ............................................................................................................................... 10

Brain-Machine Interfaces ...................................................................................................................... 10
Design Criteria ...................................................................................................................................... 12
  Wireless System ............................................................................................................................... 12
  COTS Components .......................................................................................................................... 13
  Neural Signals .................................................................................................................................. 13
Previous System .................................................................................................................................... 14

**DESIGN OVERVIEW** ....................................................................................................................... 15

Array and Connector ............................................................................................................................ 18
Bandwidth Requirements ...................................................................................................................... 20
  Compression ...................................................................................................................................... 21
  Buffering .......................................................................................................................................... 21
An Alternative Design ............................................................................................................................. 22
Amplifier ................................................................................................................................................. 23
Digital Section ...................................................................................................................................... 23
Wireless Unit ......................................................................................................................................... 26
Power System ........................................................................................................................................ 28

**AMPLIFIER DESIGN** ....................................................................................................................... 30

Filter ................................................................................................................................................... 31
Op-AMP Selection ............................................................................................................................... 33
Layout .................................................................................................................................................. 34
Noise Considerations ............................................................................................................................ 36
Prototype ............................................................................................................................................... 37

**ANALOG TO DIGITAL CONVERTER DESIGN** ............................................................................... 40

Prototype ............................................................................................................................................... 41

**WIRELESS SYSTEM DESIGN** ......................................................................................................... 43

Possible Systems .................................................................................................................................... 44
  802.11b ............................................................................................................................................... 44
  Cell phones ......................................................................................................................................... 44
  FSK module ....................................................................................................................................... 44
  Spread-Spectrum Radio Module ......................................................................................................... 45
  Bluetooth ............................................................................................................................................ 45
Bluetooth Design ..................................................................................................................................... 46
  Selection of Bluetooth Module ........................................................................................................... 47
  CSR Module ....................................................................................................................................... 48
  Testing and Implementation ............................................................................................................... 51
List of Tables
Table 1: Pinout for analog input to amplifier prototype ............................................................... 40
Table 2: Pinout of the output from the amplifier prototype .......................................................... 40
Table 3: Pinout for input from amplifier board to ADC prototype ............................................... 42
Table 4: Input and output pinouts for ADC prototype ............................................................... 43
Table 5: Basic characteristics of the CSR Bluetooth module ..................................................... 49
Table 6: Pinout for the CSR/Mitsumi Bluetooth module ............................................................. 50
Table 7: Settings necessary to reprogram the Bluetooth module for HCI mode ....................... 52
Table 8: UART packet indicators used in the Bluetooth standard .............................................. 53
Table 9: Structure of a Bluetooth HCI command packet ............................................................ 54
Table 10: Structure of an Bluetooth HCI event packet ............................................................... 54
Table 11: Structure of a Bluetooth HCI data packet ................................................................... 55
Table 12: Command sequence for setting Bluetooth master mode ............................................ 57
Table 13: Command sequence for Bluetooth slave mode .......................................................... 58
Table 14: Area estimate for subunits of the final prototype ....................................................... 64
Table 15: Power consumption of implant subunits ......................................................................... 65

List of Figures
Figure 1: Sample neural spikes recorded from monkey motor-cortex. (Jan Malášek) .................. 14
Figure 2: Robotic arm being controlled by neural signals from monkey. (Nicho Hatsopoulos) ...... 15
Figure 3: The microelectrode array prior to being cut from its base. (Tim Fofonoff) .................... 16
Figure 4: Neural waveform and trigger terminology used in digital section. (Jan Malášek) .......... 17
Figure 5: Block diagram of the implant (power system not shown). (Jan Malášek) .................... 18
Figure 6: Flexible circuit board and connector prior to assembly and connection to the array. (Jan Malášek) ............................................................... 19
Figure 7: Block diagram of the digital section showing input and output. (Jan Malášek) .......... 24
Figure 8: Organization of the small SRAM used for initial acquisition. (Jan Malášek) ............... 25
Figure 9: Buffer access sequence for the sampling section of the FPGA. (Jan Malášek) ............ 26
Figure 10: Block diagram of the inductive charging system. (Tim Fofonoff) ......................... 29
Figure 11: Reference supply and one channel of the analog front end. ..................................... 32
Figure 12: Two channel prototype amplifier using 4-channel TSSOP op-amp package. (Robert Dyer) ... 37
Figure 13: Two channel prototype amplifier using two dual-channel SOT23 op-amp ............ 38
Figure 14: Partially assembled eight-channel amplifier prototype. (Robert Dyer) .................... 39
Figure 15: Portion of the analog to digital prototype. (Robert Dyer) ........................................ 42
Figure 16: Lower part of the Bluetooth protocol stack ............................................................. 46
Figure 17: Actual size Bluetooth module prior to assembly. (Robert Dyer) ............................. 49
Figure 18: Screenshot of BlueGUI Bluetooth control software ................................................. 56
Figure 19: Bluetooth module, wire antenna, and MSP430 connected for a test. (Robert Dyer) ... 59
Figure 20: 100 μV signal generators with and without impedance control. (Jan Malášek) ........ 60
Figure 21: Primate skull showing area available for implant. (Jan Malášek) ............................. 63
Introduction

Brain-Machine Interfaces

Over the last 20 years, tremendous advances have been made in neurology in general and in the field of brain-machine interfaces in particular. New methods for measuring large-scale brain activity and new techniques for stimulating neural tissue in situ are beginning to be combined with microchip design, computer science, signal processing, and robotics. The combination of these technologies has already begun to produce results in the domain of auditory and visual prostheses. Soon many other medical devices will become possible while non-medical applications for brain-machine interfaces will begin to appear.

Nearly all, current human-computer interfaces require conscious involvement on the part of the user. Though such involvement is acceptable for certain tasks such as word-processing or drawing, it is not viable for many types of interactions. In particular, control of prostheses or “cyborg” systems is most useful if the interface is as flexible and intuitive as muscle movement. Likewise, visual and auditory prostheses are most useful if their signals reach the brain in the same way as signals from the organs they replace. In general such interfaces require a system to detect and/or induce electrical signals in the brain. The most specific connections can detect or induce signals from small groups of neurons and even individual neurons. Current recording systems use small, insulated conductors with conducting tips. The same sort of electrode can be used in reverse to induce signals at specific neural locations. Less invasive techniques such as EEG or the conductive mesh used for epilepsy research do not currently allow the detection or manipulation of signals from small groups of neurons.

Cochlear implants are the first brain-machine interface to be widely used in humans. These auditory prostheses work by converting acoustic signals into patterns of electrical stimuli that are fed into an array of electrodes implanted in the patient’s cochlea. These electrodes deliver their signals to auditory nerve fibers in a way that simulates the signals produced by normal audition. Tens of thousands of deaf patients have had their hearing
partially restored by these devices. Though vision prostheses have proven somewhat more difficult to design, approaches similar to those used in cochlear implants show promise.

Brain-machine interfaces can also be used to control undesirable brain activity. This is true because direct electrical-stimulation of brain tissue can disrupt certain pathological patterns of brain activity such as those associated with Parkinson's disease, epilepsy, and some forms of chronic pain. The current systems used for treatment of Parkinson's disease depend on open-loop control: the stimulating signal is adjusted until the symptoms are mitigated and must be adjusted by hand over time. Epilepsy control systems will probably need to be closed loop: the system must detect the onset of a seizure and mitigate it via electrical or chemical intervention. Such closed-loop systems have been demonstrated to work in rats and undoubtedly represent one form of brain-machine interface that will be used in humans very soon.

Brain-machine interfaces will one day make the idea of a cyborg perfectly natural. They will allow an amputee to control a prosthetic limb or retransmit signals around a spinal injury, allowing an individual to regain control of a paralyzed limb. Artificial cochlea and artificial retina represent the leading edge of brain-machine interfaces that enhance perception. More complicated systems may one day provide their users with new forms of sensory input such as radar or infrared vision. Likewise, such interfaces may one day allow subconscious control of vehicles or remote drones. At a clumsier level, multichannel EEG has already been used to give quadriplegics control over computer cursors.

In the near term, the creation of such interfaces demands basic research in neurology in addition to advances in technology. The goal of the Telemetric Electrode Array System (TEAS) project is to create new hardware that will serve as a prototype for the embedded electronics necessary to build a brain-controlled prosthetic device. To maximize the utility of such a device it must be able to detect the relevant neural signals with minimum noise and damage to the surrounding delicate brain tissue. This hardware will also
facilitate neurology research that may someday lead to better brain-machine interfaces. Our implant is designed to record up to 64 channels of neural data using 64 electrodes inserted in the motor-cortex of a small, non-human, primate. The data acquired from the motor cortex will be used to develop better models relating neural signaling to limb movement. These models could then be used to develop prosthetic devices for humans.

**Design Criteria**

**Wireless System**

The system currently used by our colleagues at Brown University requires that the subject be attached to an acquisition system by a cable harness\(^2\). This cable harness has one wire per electrode in addition to wires for reference and ground. The signal from the electrode can be buffered between the transcutaneous connector on the subject's head and the cable harness. The cable harness connects to an amplifier and filter bank. Each signal is then digitized and made available for analysis. The use of the cable harness presents several problems. Its long length, and the lack of significant amplification at the subject's head make the signals traveling through the harness (which may be several meters long) vulnerable to electrical interference. In some situations the experimental subjects may also chew or otherwise abuse the cable\(^3\). Obviously, such a large cable harness is not viable in a prosthetic device designed for humans. In addition, the transcutaneous connector presents a source of infection for the subject as well as an area that must remain clean when the system is not in use. One solution to all of these problems is the use of a wireless link to transmit the neural data. Such a link circumvents almost all the problems associated with a cable harness, though care must be taken to avoid creating additional, radio-generated, electrical interference inside the implant. A wireless link also makes new types of experiments possible as it removes the need to immobilize the subject during experiments. In order to use a wireless link, it is necessary to do a significant amount of data processing. The signals from the neurons must be amplified, filtered, digitized and prepared for error-free transmission\(^4\). This processing must be done inside the skin of the subject if all the benefits of using wireless transmission are to be realized.
COTS Components

The current version of TEAS uses commercial electronic parts for data processing. Though integrated circuits designed specifically for this application would potentially have offered superior performance, the use of commercial off the shelf (COTS) parts allows for quick design of a working system based on well-specified parts. COTS parts, specifically designed for low-power operation and optimum performance, are at least comparable, if not superior, to custom designed parts\(^4\). However, custom chips would have offered significantly greater levels of integration and thus allowed for the creation of a smaller implant. In particular, the analog front end for the implant could be made significantly smaller if a custom analog circuit were used.

Neural Signals

Current theory indicates that most, if not all, information transmitted between neurons is encoded in the timing between neural spikes\(^2\). The shape of individual spikes can be used to differentiate between neurons and may be of interest in neurological research but isn’t used in interpreting motion. An electrode in the system detects a spike train from one or more neurons close to its tip. Each spike lasts one to two milliseconds and has a peak-to-peak amplitude on the order of 100 microvolts\(^2\). Figure 1 shows some typical neural spikes.
Each spike in a spike train corresponds to its associated neuron firing. Typically neurons in the motor cortex fire at rates between \(10\) Hz and \(300\) Hz. Faster firing rates correspond to more rapid muscle movement.

Though some research and control problems may require thousands of channels of neural data, current research indicates that 30 to 100 channels of neural signals from the motor cortex are sufficient to reproduce the three dimensional motion of a limb\(^6\).

**Previous System**

Our system will initially be used in experiments at Brown University. These experiments are part of a primate study that aims to correlate signals in the motor cortex with arm motions. The subjects, typically Macaque monkeys, are trained to play a computer game: when they move a mouse-controlled cursor onto a target they are rewarded with a sip of juice, and the target then moves to another position. An electrode array is then surgically implanted in the subject's motor cortex. Transcutaneous connectors provide connections to the electrodes as well as reference wires. The reference wires are stainless steel,
flexible, insulated wires; a few inches of the insulation is stripped at the end and the bare wire lies on the surface of the motor cortex to provide a voltage reference for the amplifiers used to amplify the signals from the electrodes. After this surgery, signals from the motor cortex are recorded simultaneously with a record of cursor movement during game play. Analysis of these two data sets provides the models needed to predict the movement of the cursor (and thus the monkey's hand). The mouse input can then be disconnected from the system and the models used to allow the subject to play the game. The signals from the subject's motor cortex control the cursor on the screen. These same models can also be used to control a robotic arm that will mimic, to some extent, the motion of the subject's arm. Figure 2 is a picture of a robotic arm being controlled by signals from a monkey's motor cortex.

![Figure 2: Robotic arm being controlled by neural signals from monkey. (Nicholas Hatsopoulos)](image)

**Design Overview**

The TEAS implant is made up of five major submodules: the electrode array and connector, the analog front end, the digital section, the Bluetooth radio module, and the
power system. The 8 by 8 electrode array is made of stainless steel or titanium. It is the sensor for the system; each electrode is electrically insulated except at its tip, which is platinum coated. Each electrode is connected to a separate analog channel by a trace on a flexible printed circuit board (PCB)\(^5\).

Each channel of the analog section consists of an amplifier and a filter. The signal from the second amplifier is then fed into one input of an analog-to-digital converter (ADC). Each ADC multiplexes eight analog signals, digitizes them, and outputs a single serial data stream to the digital section. The digital section is built around a field programmable gate array (FPGA) and a microcontroller. The FPGA acts somewhat like a digital oscilloscope: data from each channel are written to an output buffer when the signal from that channel passes a threshold (i.e. the channel triggers). A small ring buffer acts as a prestore for each channel and thus makes available samples from before the trigger. The poststore data-points are acquired directly, after the channel triggers (Figure 4).
The microcontroller is essentially an interface between the FPGA and the Bluetooth radio module. It also serves as a programmable power manager and controls the overall high-level behavior of the implant. The Bluetooth radio module is the implant’s only communication link. The implant’s power system must be small, while providing sufficient time between recharge cycles, fast recharge times, and sufficient stability to allow for high precision measurement. The power system includes independent regulators for various parts of the circuit, a lithium polymer battery, and an inductive charging circuit that allows recharging of the battery through the skin of a subject. A block diagram of most of the implant appears below (Figure 5).
Figure 5: Block diagram of the implant (power system not shown). (Jan Malášek)

Array and Connector

Most of the work involving the array was done by other members of the group. This section will give an idea of the issues involved in its design and construction. The 8x8 electrode array is manufactured out of a piece of stainless steel using wire electrical discharge machining (EDM). The piece is machined twice, rotating the work-piece 90 degrees on the second run so that an array is produced. The array is then electroplated with platinum for conductivity and biocompatibility. A square piece of polyimide with 64 holes drilled in it is epoxied near the bottom of the array. Then the base of the array is cut off using EDM to electrically isolate the electrodes, leaving short stubs protruding from the back of the polyimide square.

The array is connected to the module electronics by a flexible printed circuit board and an 80-pin microconnector. The 64 traces that connect the solder pads at the array end to the electronics are etched on a sheet of flexible printed circuit board (PCB) material. The traces are connected to the electronics by a fine-pitch surface-mount connector. A picture
of the connector and flexible circuit board appears in Figure 6. The end of the flexible board fits over the array stubs and is hand-soldered into place. The entire array and connector are then coated with parylene insulating coating for biocompatibility and electrical insulation, leaving only 50 μm at the tip of each electrode exposed.

The connection between electronics and array must be flexible and malleable. Flexibility allows the electrodes to move with the brain as it moves relative to the cranium. This movement can be on the order of millimeters. The flex circuit board provides fairly good flexibility in one plane since it will bend up and down easily. Additional flexibility is achieved by laser-machining slots between each of the traces. During implantation, the flex circuit will be attached to the array but not the electronics to allow the surgeon to...
place the array as easily as possible. The connector must be malleable so as to allow the neurosurgeon to properly position the array during surgery, prior to inserting the array into the motor cortex with a pneumatic hammer\textsuperscript{5}.

Two coatings are required for the electrode array: a conductive layer and an insulating layer. Both layers need to be biocompatible, as they will be in direct contact with the neural tissue and the corrosive cerebro-spinal fluid (CSF). The conductive coating is necessary to maximize signal pickup and minimize chemical interactions between the electrode tips and the CSF. Gold, platinum, and titanium nitride were investigated as possible coatings. Titanium nitride has similar conductive and biocompatible properties to gold and platinum; however, it is harder to apply. Gold and platinum are fairly similar in conductivity and biocompatibility properties. We chose platinum, which has been used successfully in other electrode arrays. The platinum layer was electroplated onto the electrode array.

The insulating coating, which covers most of the electrodes, allows conduction only through the tips. This insulation is essential for isolating signals from single neurons or small groups of neurons in close proximity to a tip. The area and geometry of the exposed section of tip, as well as the tip material, controls the electrical impedance of the electrode and its signal pickup characteristics. The insulation is a biocompatible polymer coating: parylene. This coating can be applied in vapor form in order to evenly coat the irregular surfaces of the electrode array\textsuperscript{4}.

**Bandwidth Requirements**

It is possible to predict muscle movement by looking only at inter-spike timing from a small number (30 to 100) of channels\textsuperscript{10}. However, for research purposes and for distinguishing between spikes from different neurons on the same channel, it is essential to be able to examine the shape of the spike. Unfortunately, the amount of bandwidth necessary to transmit digitized versions of all 64 channels in the implant is enormous: 64 channels × 12 bits × 31.25 kHz = 24 Mbits/s. This calculation ignores communication overhead and it still gives a bandwidth far in excess of commercially
available, low-power radios. If one assumes that the typical maximum firing rate of a channel is 300 Hz\(^2\), and one only collects samples around a trigger threshold (as is often done in neural recording) then the bandwidth requirement drops to:

\[
\text{64 channels} \times \text{64 samples} \times \text{12 bits} \times 300 \text{ Hz} = 14.7 \text{ Mbits/s.}
\]

Again this is too fast.

**Compression**

Compressing the information about the spikes might allow one to reduce this bandwidth requirement. Unfortunately, a preliminary analysis of the compressibility of spike data is not promising. One simple method that would allow for some compression is to encode the stepsize between successive samples as opposed to the actual sample points. Such delta compression shows some promise: in Matlab experiments on recorded neural spikes it seems to reduce the effective number of bits needed to encode a sample from 12 to 10 or so. Delta compression would seem easy to perform in hardware; however, it requires that each channel’s data be kept separate until after delta compression. Delta compression becomes useless if it is applied to the stream of sample points obtained after multiplexing the samples from each ADC, as successive sample points would then be essentially uncorrelated. Keeping each data stream separate would introduce significant complications into the design of the digital section of the implant. There don’t seem to be any easily applicable data compression techniques that significantly reduce the size of the multiplexed bitstream. Lempel-Ziv compression\(^36\), for example, gives no significant improvement. More complicated approaches, such as fitting curves to the spike data, require that data from each channel be stored and compressed separately; again this level of complexity is probably not viable in our current system.

**Buffering**

Buffering signals would reduce the demand on bandwidth by allowing the transmission bandwidth to be closer to the speed determined by the average neural spiking rate of 10 Hz\(^2\): 64 channels \times 64 samples \times 12 bits \times 10 \text{ Hz} = 491 \text{ kbits/s.} This bandwidth is easily achievable with various commercial technologies. However, buffering reduces the real time nature of the data. Fortunately, most of the current research situations in which seeing the shape of spikes is desirable do not require the data to be available in real-time.
In fact, the only scenario in which the data need to be available in near real-time is prosthesis control; at the moment, simple inter-spike timing data seems to be sufficient for such applications. Spike-timing data can be transmitted over a relatively restrained bandwidth: 64 channels × 300 Hz × 16 bit timestamp = 307 kbits/s. Again, a little buffering would dramatically reduce this requirement because the average firing rate is lower. The timing data may be more amenable to compression: as a first cut, a shorter time-stamp could be used in conjunction with a more frequent time-announcement message or by simply depending on the receiver detecting time-stamp wrapping. More complicated approaches would compress information about spikes that occur relatively close together in time.

An Alternative Design

An alternative design would eliminate the FPGA and microcontroller and replace them with a much higher speed microprocessor. The StrongARM class of microprocessors, for example operates at speeds in excess of 200 MHz; some DSP chips might also be suitable. This is fast enough to handle the serial interface to one or two higher speed analog to digital converters as well as triggering and memory interfacing done by the FPGA in the actual design. The microprocessor would also easily reproduce the functionality of the microcontroller. This design would seem to reduce the part count and concentrate the software design of the module into a single device. However, the additional support circuitry needed for most high-speed processors (RAM, startup hardware, etc.) nearly negates the size advantage. The two designs are comparable in their power requirements. The FPGA design is more flexible and more scalable. The current design could be scaled up to handle twice as many channels without any significant hardware modification by replicating the VHDL code for the FPGA. This simple scaling by replication is impossible in a microprocessor based design. The use of an FPGA also gives us more flexibility to implement some signal processing, compression, or encoding at the hardware level.
Amplifier

Single-supply op-amps are used to minimize size, power-consumption, and the complexity of the implant’s power supply. In order to use such op-amps it is necessary to bias the inputs from the electrodes. This bias voltage ($V_{bias}$) is generated by buffering half the ADC’s reference voltage. Thus, all of the analog signals in the system will be centered on the middle of the ADC’s range. The reference electrode is a low-impedance connection to the area in which signals are being collected. This electrode effectively biases the DC level for the neural spikes in the motor cortex. The reference could be created with a set of low-impedance electrodes on the array, but it is easier to use a set of wires resting on the surface of the brain. A resistor-capacitor pair between the two amplifier stages serves as a first-order high-pass filter. This filter blocks DC offsets in the incoming signal as well as those introduced by the first stage amplifier.

Digital section

The digital section reads samples from the ADCs, compresses the data by only capturing samples around a trigger point, and sends that relevant data to the Bluetooth module. As shown in the block diagram in Figure 7, the digital subsystem has four primary components: a programmable logic device for doing high-speed sample analysis; a small SRAM to hold pre-store samples prior to triggering; a microcontroller to interface to the Bluetooth module and to control the whole implant; and a large SRAM buffer for holding data that is ready to be sent. The digital section is described in greater detail elsewhere.
The programmable logic device reads samples from the ADCs, determines which samples are relevant, and schedules them for transmission. These three tasks are performed in three distinct modules on the FPGA; there is also a counter module that synchronizes the other modules and provides a clock for time-stamping data. The 12-bit ADCs require sixteen serial clock cycles to transfer out a conversion; the timing on the eight ADCs is staggered so that a new sample is available every two cycles. The serial clock is half the frequency of the 8 MHz system clock, and since two serial cycles are needed per sample, the aggregate sampling rate is 2 MHz. The triggering system, which runs at the system clock speed, thus has four cycles to process each sample.

The triggering module compares a new sample from a channel to a set of triggering criteria, if the sample exceeds the trigger level the prestore for the channel is transmitted along with subsequent (poststore) data. This module must solve a context switching
problem arising from the interleaved arrival of samples: after a reading from a given electrode is processed, 63 other samples will be processed before receiving the next sample from the first electrode. Furthermore, each channel must have its own pre-store buffer that holds the most recent samples measured on that channel. The memory requirements cannot be met by the few bits available in the FPGA, so a small, 4Kx14 external SRAM is used. The memory organization is shown in Figure 8. Each channel uses 64 words, of which four have special values and the rest of which are available for storing samples. A status word holds all information about the state of the channel, such as whether or not it is triggered or how many samples have already been read. The other special values hold the trigger value, the number of samples to be read after triggering, and an un-trigger value that provides hysteresis.

![4Kx14 Sampling Unit Memory Organization](image)

Figure 8: Organization of the small SRAM used for initial acquisition. (Jan Malášek)

Of the four clock cycles available for each sample, two are spent reading and writing back the status value. The remaining two cycles are used to either read a threshold and store a sample, or to read two samples. After a channel triggers and stores the correct number of post-store samples, it dumps the samples to the large transmit buffer. No samples can be taken during the transfer time, but the duty cycle of neural signals is less than 50%, allowing ample time after triggering for transferring the data. The ability to read two samples in one four-cycle sample period allows the buffers to be cleared twice as quickly as they are filled, so that if a 1.5 ms spike is recorded, there is a 0.75 ms dead time during which the data are transferred out of the ring buffer and into the transmit buffer. The timing of the memory accesses to the ring buffer and transmit buffer are shown in Figure 9.
The transmit buffer is a large FIFO buffer that allows recording of bursts of neural activity that occur too rapidly to be sent out over the Bluetooth link. The number of samples that are sent per spike can be varied, and one feature is a timestamp only mode, in which only the times at which spike occur are sent; in this mode, firing rates of up to 300 Hz on all 64 channels can be observed in real time over the Bluetooth link.

The FPGA used for the TEAS digital system is the Altera FLEX 10K30A, which has sufficient gate and I/O counts to implement the design. The Altera device has relatively low power consumption, and it is available in an fine line ball grid array (fBGA) package that measures 17 mm on each side. Because it is SRAM-based, however, the device must be configured every time it is powered up; the microcontroller, which has spare FLASH memory available, is used for this purpose to avoid adding components to the design.

**Wireless Unit**

A commercial Bluetooth module from Cambridge Silicon Radio (CSR) provides the wireless link for the module. The Bluetooth protocol is an international standard for the wireless transmission of data over short distances. It uses frequency hopping spread-
spectrum (FHSS) techniques to minimize interference between adjacent Bluetooth devices as well as to provide some protection from ambient electromagnetic noise. The Bluetooth protocol was originally designed to allow wireless connectivity between small, low-power consumer electronics products. Thus, the Bluetooth radios available at this time have been designed with size, interference, and power constraints in mind. Therefore, Bluetooth radios provide a relatively convenient COTS solution to the problem of wireless communication. The Bluetooth protocol specifies several levels of interaction between a user and the radio. Most modules hide the lower levels and present a generic high-level interface called the host controller interface (HCI) level. It is possible to control a Bluetooth module over a universal asynchronous receive/transmitter (UART) or universal serial bus (USB) interface at this level with relatively simple commands. These commands allow interrogation of adjacent Bluetooth units, connection setup and error-correcting data transfer without needing to understand or explicitly control the FHSS radio. Once a connection is established, the radio module can be used very similarly to a normal serial connection. The size, noise-immunity, efficiency, and ease of use of Bluetooth modules makes them one of the best options among commercial radio modules.

The current Bluetooth standard calls for a transmission rate of 725 kbits/s\textsuperscript{21}. In order to take advantage of the maximum speed of the modules the communication between the implant's microcontroller and the Bluetooth module must be faster than 725 kbits/s. Since the Bluetooth module is the only communication link with the outside world, the communication protocol must also provide a path for commands controlling the rest of the implant. This functionality includes selecting channels, setting thresholds in the sampling section, setting sampling speed, initiating recharging cycles, and any other commands related to power management. The hardware for the external transceiver will essentially duplicate the Bluetooth radio in the implant. However, since the transceiver will be connected to a personal computer some modifications of the radio's interface will be necessary. In order to ensure an adequate bandwidth between the controlling PC and the Bluetooth module, a fast (1.3 MHz) serial card is necessary. It would also be possible to replace the serial interface with USB. The control software for the system will
provide an interface for selecting channels for recording, setting channel thresholds, changing the recording mode (e.g. real-time, buffered, time-stamp only), and controlling higher level implant functionality such as sleep mode or charging. The control software will store the captured data in formats suitable for analysis. This software could be extended to provide real-time analysis of signals.

**Power System**

Power management is a critical aspect of the design. The entire system must operate off a surgically implantable battery. With this power source, the system must provide the functionality of a wireless, 64-channel oscilloscope for several hours. The system also provides a low-power sleep mode, during which the battery can be inductively charged. Though it may be possible to run the system while it is being charged, noise considerations may make this impossible. Such “plugged-in” operation would also require the charging circuit to provide enough current to operate the system. For maximum energy density, we will use rechargeable Lithium-polymer batteries. These batteries have energy densities on the order of 500 kJ/kg. A small battery should be able to power the implant for two to three hours. Lithium-polymer batteries are less prone to overheating or exploding during charging than Lithium-ion batteries. However, they still present some significant obstacles to safe, inductive charging. A charging control IC is required to recharge these batteries safely. Because the supply for this IC will be an inductive pickup, the charging IC must be able to function correctly in the face of supply interruptions and noise. A sketch of the system appears in Figure 10.
The external coil induces a current in the internal pickup. The resulting voltage is rectified and applied to a filtering capacitor. The voltage across this capacitor is the supply for the charging IC. This supply is somewhat noisy, but any regulation would be costly from a size and power perspective. Since the external coil will be positioned on the subject with a belt or jacket connected to a power source, it is likely that the inductive power transfer will occasionally be interrupted as the subject moves or the power supply is disconnected. Therefore, the IC must also charge the batteries correctly despite power interruptions that could last from milliseconds to minutes. This requirement disallows charging-control ICs that depend on timing to determine the end of certain charging periods.

The microcontroller and Bluetooth link must remain on continuously. However, both of these parts have low-power sleep modes. All other parts of the implant can be turned off by the microcontroller (in response to a command sent over the radio link). This shutdown feature is implemented by cutting power to submodules in the implant. This control is made possible by using separate linear regulators with shutdown inputs, though a single regulator with switched outputs could also work. The separate regulators provide some added isolation between modules such as the analog amplifiers and the FPGA.
Amplifier Design

Each channel of the analog front end must amplify very small signals (100 $\mu$V peak-to-peak), introduce minimal distortion and noise, serve as an antialiasing filter for the ADC, and present a high impedance input to its corresponding electrode. Since any circuit designed for this purpose will be replicated 64 times (once for each sensor electrode in the array), the circuit must be physically small and consume minimal power. Most of these goals can be met by using CMOS-based operational amplifiers (op-amps). CMOS technology offers low power, small size, and reasonable performance characteristics. One of the most important of these characteristics is the gain-bandwidth product (GBW) of the amplifier. Since we desire a bandwidth of approximately 10 kHz and a gain of about 10,000, a single op amp would require a GBW of 100 MHz. Unfortunately, small, low-power op amps tend to have GBWs lower than 15 MHz. One solution is to create a two-stage amplifier.

Previously published designs for array front-ends have used dual supply op-amps\textsuperscript{14} because most high performance amplifiers demanded dual supplies. Since these front-ends were not designed to be implanted, the added complexity of a second power supply was not a significant problem. However, in the TEAS implant, a second power supply means the inclusion of an additional battery or bias circuit as well as more power conditioning circuitry. High-end single-supply op-amps have smaller supply voltages than equivalent dual-supply chips: thus, for equivalent supply current, single-supply amplifiers consume less power. The design of a single supply amplifier circuit is only slightly more complicated than that of a dual supply circuit. In order to accommodate positive and negative going signals the op-amps in the circuit must be biased around a “middle voltage” this bias voltage must also be used as the reference for the neural spikes in the brain. This reference/bias voltage should also match the middle-point in the analog to digital converter’s range. Two high-value resistors in a divider circuit provide the original reference. A simple op-amp buffer provides the necessary (rather minimal) current requirements for the “reference supply”.
CMOS op-amps have significant input offset voltage (typically over 1 mV). If the op-amp is being used in an amplifier, the offset will appear at the output, scaled by the gain of the amplifier. Since the TEAS amplifiers acquire signals that are approximately ten times smaller than this offset, it is necessary to introduce a circuit that will block the effects of the offset. Fortunately, the offset varies slowly, and the spike information does not contain low-frequency information. It is relatively easy to block the DC components of the amplified signal prior to passing the signal to the second amplifier stage. If the first stage gain is well chosen, the offset voltage of the second stage is not a problem because the output signal from the first amplifier is significantly larger than the offset voltage.

**Filter**

A simple RC high-pass filter is sufficient for DC-blocking. The filter was designed to match the filtering done by our collaborators’ current acquisition system. It isn’t necessary to match this filter since additional filtering can be done after spike acquisition. However, the current filter has proved to be useful so it is more convenient to do so. This filter’s 3 dB point is 200 Hz. This cutoff frequency also has the advantage of attenuating 60 Hz noise in the system. The value of $R$, the resistor in this filter, must be chosen so as to match the filter capacitor (whose value is restricted by the availability of very small capacitors). $R$ must not be so large as to introduce significant offsets due to the bias current of the op-amp. Minimizing the bias current of the amplifier would extend the range of usable parts.

Low op-amp bias current is also significant in avoiding charge saturation of the electrodes and associated voltage drift in the measurement. Charge saturation can also be reduced by making the return path impedance as low as possible. Fortunately, the reference wires used in making the measurement also provide a low impedance return path.
The low-pass antialiasing filter for this circuit is provided by the frequency-gain response of the op-amps. The desired 3dB point of this filter and the GBW of the op-amp set the maximum gain of the two amplifiers: 
\[ gain = \frac{GBW}{f}. \]
Our collaborators' current system has a low pass filter with a cutoff around 12 kHz. The current TEAS amplifier has a 3db point around 10 kHz. The shape of this filter's rolloff is set by characteristics of the op-amp. The gain of the amplifier is set by taking the desired overall gain and dividing it by the gain necessary for the antialiasing filter. The larger gain is best implemented in the second stage amplifier since it has much lower impedance at its input and thus any offset due to bias current will be minimized. Figure 11 shows the complete schematic for the amplifier and the reference supply. Note that the reference supply does not need to be duplicated 64 times.

![Diagram of the circuit](image)

**Figure 11:** Reference supply and one channel of the analog front end.

The analog front-end is one of the largest submodules in the implant. The amplifier and filter portion of the circuit in Figure 11 must be reproduced 64 times in the implant. If the two op-amps are contained in a single package, this translates to \(7 \times 64 = 448\) parts in the amplifier portion of the analog section. This number does not include any decoupling capacitors that may be necessary. It is primarily for this reason that more complex filters cannot be used for the high-pass and antialiasing filters of the front-end. The simple, first-order filters in the TEAS system represent the best trade-off between size and
performance for the front-end. Given the small size of the subject’s skull, it will be necessary to use the smallest surface-mount components available (e.g. 0201, TSSOP, SOT23).

**Op-amp Selection**

As described above, the most significant performance parameters for the op-amps used in the analog front-end are the frequency response versus gain (i.e. the gain bandwidth product), the input impedance, power consumption and the overall size of the op-amp package. Noise characteristics, offset voltage, drift, and power supply requirements are also important parameters; however, op-amps that met the first set of requirements by and large had acceptable secondary characteristics. Temperature drift is even less important as the subject’s cranium stays at a relatively constant temperature. The overall gain of the amplifier circuit needs to be approximately 10,000. A gain of 10,000 will amplify a typical 100 μV neural spike into a 1 V peak-to-peak spike. This gain allows spikes with three times the typical amplitude to be amplified without distortion and also makes smaller amplitude spike visible. Since the spikes are centered in the center of the ADC’s range, this gain also allows for some drift in the system without saturating the amplifiers or the ADC. Commercial, bench-top data acquisition systems allow the user to individually adjust channel gains. Such a feature might be possible to implement but at the cost of significantly greater circuit complexity, size, and power requirements. The overall gain for the system must be split between the two amplifiers. The second amplifiers gain must be larger than the first and chosen so as to create the system’s anti-aliasing filter. The minimum acceptable GBW is set by the cutoff frequency of the filter, 10 kHz, and the square root of the desired channel gain:

$$GBW = f \cdot \sqrt{Gain} = 10kHz \cdot 100 = 1MHz.$$  

The input impedance of the first-stage op-amp determines the offset voltage due to the loading of the electrode. This loading can also lead to charging of the electrodes that introduces additional drift into the system. Typical intra-cortical recording electrodes have impedances around 100 kΩ at 1 kHz and 2 MΩ at DC. Fortunately CMOS op-amps
usually have input impedances in the giga to tera Ohm range. Bias current is closely tied to input impedance but usually better characterized. It is significant for the same reasons as input impedance.

The power supply requirements for the op-amps have been partly discussed in the section describing trade-offs between single and dual supply op-amps. In order to simplify the design it would be best that the op-amps operate at the same supply voltage as the digital section. Due to various constraints in the digital section, the only useful supply voltage is 3 V. Many low-power, single supply CMOS op-amps can operate at this voltage. A related characteristic of op-amps, which can be significant in the selection, is the input and output range. Since the input to both op-amps is essentially the bias voltage, it is only necessary to insure that the input range extends past VDD/2 (1.5 V). The differential mode voltage of the inputs is essentially zero so this occasionally frustrating characteristic of op-amps is not relevant. It is desirable that the second op-amp have as wide an output range as possible, preferably rail-to-rail. A full-scale output swing extends the dynamic range of the amplifier-ADC circuit.

The OPA237 op-amp\(^\text{17}\) stands out in large part due to its high GBW and its small size dual SOT23 package. It also has the advantage of being available in small quantities.

**Layout**

Small op-amps are typically manufactured in TSSOP and SOT-23 packages. TSSOP packages usually offer two or four channels, and SOT-23 packages are usually dual or single packages. The SOT-23 dual channel packages are less than half the size of the TSSOP packages; however, the density of the two packages is fairly comparable since each power distribution to two dual packages is more complex than that for a single four-channel device. If it were necessary to decouple each amplifier, the SOT23 packages would lead to a larger layout; however, a single decoupling capacitor can be shared between several amplifier circuits. If one uses dual channel op-amps, there are two layouts possible: implementing the two-stage amplifier for one data channel in a single package or implementing two data channels across two amplifiers. Sharing packages
between amplifiers makes the layout slightly simpler but introduces the possibility of cross-talk between channels occurring inside the op-amp. The single-package approach is a bit more complex, but any cross-talk between amplifiers would be confined to the same channel and would not present a problem. Four-channel TSSOP packages offer the same sort of trade-offs. The OPA237 is available in a dual-channel SOT-23 package. The current prototype implements each channel of the amplifier in a single package. No problems with internal cross-talk have been noted. The layout of a channel is smaller than prior layout based on four-channel op-amps.

Given the very low voltage signals being measured, it is essential that the amplifiers be as immune to noise as possible and not introduce excessive noise themselves. Though decoupling each amplifier from the power rails would be ideal, the number of additional components (at least 64) is prohibitive. Instead, careful separation of the power planes for the amplifier section from those used by the ADCs, digital section, and radio as well as limited numbers of decoupling caps are combined to protect the analog front-end as much as possible. The power planes for the entire system are connected in a “star” configuration with only one connection as close to the power input point as possible. This arrangement reduces the impact of the noisy digital power planes on the analog section and reduces the possibility of ground loops. Decoupling capacitors will be distributed among groups of amplifiers. At the moment, each group of eight amplifiers (associated with an ADC) has one set of decoupling capacitors.

Since each dual op-amp implements one amplifier, it is relatively easy to route signals so as to avoid excessive cross-talk. High-voltage (amplified) signals do not need to be routed close to low-voltage (input) signals. Cross-talk between signals of comparable amplitude does not appear to be an issue at the relatively low frequencies involved. In particular, no provisions were made to isolate the input signals from each other by interleaving them with ground lines.
**Noise Considerations**

The analog front-end circuitry can be a significant source of noise in and of itself. The amplifier's intrinsic voltage and current noise as well as noise due to leakage currents on the PCB all contribute to this "self-noise." Voltage noise appears as noise on the input offset voltage of the op-amp. Current noise is noise in the bias current. The current noise is particularly troublesome given the high impedance of the electrodes. Careful selection of op-amps is the only way to minimize voltage and current noise; fortunately, CMOS amplifiers typically have low current noise. Leakage currents through the dielectric material of the PCB or through surface contaminants and into amplifier inputs can also produce noise. In addition to thoroughly cleaning the PCB to remove flux residue and other contaminants, this effect can be mitigated by placing guard rings around input pads and traces. These guards are traces connected to ground that will absorb any stray currents in the vicinity of the input pads. Such leakage currents do not seem to be a problem in the TEAS prototypes; therefore guard rings, which are expensive in terms of layout space, have not been used in the current design.

Electromagnetic interference (EMI) from external sources is probably a more significant problem than cross-talk and self-noise. Current systems for acquisition of neural data suffer from a wide variety of EMI problems including 60 Hz hum, burst noise probably related to digital wireless transmissions, and microphonic effects in connectors\(^3\). Microphonic effects occur when connectors that carry neural signals from the subject are vibrated by the subject's vocalization. Low pass filtering of the resulting signals eliminates most of the resulting noise but does not eliminate signals in the audible range: it is possible to hear a subject if a neural channel is passed into a speaker. Microphonic effects should be reduced in our system since the connectors are glued into place prior to embedding the device. Experimentally, the strength of microphonic and other interference is closely related to the existence of long grounding paths, faulty grounds, and ground loops. Our system is physically small (reducing the length and impedance of ground paths), battery powered, and embedded within the subject; all of these factors should make the device, and particularly the analog front-end, less vulnerable to EMI than existing, non-imbedded systems.
Prototype

The original prototype for the analog front-end used a four channel, dual-supply op-amp in a TSSOP, four-amplifier package. Each channel used two amplifiers in a package. A picture of this prototype appears in Figure 12. The resistors used in the prototype are 0201 packages in order to better approximate the behavior of the final system.

![Two channel prototype amplifier using 4-channel TSSOP op-amp package. (Robert Dyer)](image)

This prototype worked adequately in conjunction with the analog to digital converter; however, it did exhibit a tendency to saturate or wildly oscillate when the input was left floating. The TLC226 op-amp used in the first prototype was replaced by the OPA2337 whose characteristics were deemed superior. A picture of this later prototype appears in Figure 13. The OPA2337 comes in a dual amplifier SOT23 package. The current prototype uses one package per amplifier channel.
Figure 13: Two channel prototype amplifier using two dual-channel SOT23 op-amp packages. (Robert Dyer)

Figure 14 is the eight-channel amplifier board used for system level testing of the amplifier, ADC, and digital section. It was designed to interface to any of the eight analog to digital converters on the ADC board through a single ribbon cable. This cable carries power and reference voltage in addition to the eight channels of amplified neural signals.
The header pins at the bottom are used to connect to electrodes for testing of the amplifier with real neural signals. Table 1 shows the pinout for this connector and Table 2 shows that of the ADC connector.
Table 1: Pinout for analog input to amplifier prototype.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Electrode In</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Electrode In</td>
</tr>
<tr>
<td>Reference</td>
<td>Electrode In</td>
</tr>
<tr>
<td>Reference</td>
<td>Electrode In</td>
</tr>
<tr>
<td>Reference</td>
<td>Electrode In</td>
</tr>
<tr>
<td>Reference</td>
<td>Electrode In</td>
</tr>
<tr>
<td>Reference</td>
<td>Electrode In</td>
</tr>
<tr>
<td>Reference</td>
<td>Electrode In</td>
</tr>
</tbody>
</table>

Table 2: Pinout of the output from the amplifier prototype.

<table>
<thead>
<tr>
<th>Power</th>
<th>Analog Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>Analog Out</td>
</tr>
<tr>
<td>Reference</td>
<td>Analog Out</td>
</tr>
<tr>
<td>Reference</td>
<td>Analog Out</td>
</tr>
<tr>
<td>Ground</td>
<td>Analog Out</td>
</tr>
<tr>
<td>Ground</td>
<td>Analog Out</td>
</tr>
<tr>
<td>Ground</td>
<td>Analog Out</td>
</tr>
<tr>
<td>Ground</td>
<td>Analog Out</td>
</tr>
</tbody>
</table>

**Analog to Digital Converter Design**

In a system designed to only detect spike trains and not record the shape of individual neural spikes, an analog to digital converter would not be necessary. A comparator with a programmable threshold would be sufficient. The TEAS system does need to record spike shapes and thus requires an analog to digital module. The first decision to be made
in the design of the ADC section is the topology of the circuit. The input is 64 analog channels from the amplifiers. The output needs to be digitized information suitable for analysis and triggering in the digital section's FPGA. Some sort of multiplexing is obviously necessary. One possibility is the use of analog multiplexers ahead of the actual analog to digital conversion. An alternative is the use of ADC chips with built in multiplexers. Using eight eight-to-one multiplexers in cascade with an eight-to-one ADC requires nine ICs. Other configurations would allow one to use two or four ADCs instead of one, reducing the speed requirements on the ADC. A design using eight ADCs with built in multiplexers offers the lowest chip count and lower speed requirements on the individual ADCs at the cost of a wider databus to the FPGA. This cost is minimized if the output from each ADC is a single serial line.

The TEAS analog to digital section must have at least twelve bits of resolution, partly to match the specifications of existing systems\(^3\) and partly to reduce the amount of gain necessary in the amplifier stage. In our three volt system, twelve bits corresponds to a voltage resolution of approximately 1 mV, which is more than adequate to reproduce the shape of a 1-Volt peak-to-peak neural spike. The sampling frequency per channel needs to be in excess of 20 kHz\(^3\). This frequency requirement translates to 160 kHz for an eight-channel ADC.

As for the op-amps, the power and voltage supply requirement of the ADC are significant. The ADC must operate at 3 Volts with minimum power requirements, especially if eight or more ADCs are used. The control interface should require a minimal number of lines from the FPGA; ideally, the interface would be serial and only require two lines.

**Prototype**

The MAX1281 ADC was chosen for its relatively low power requirements, its eight-to-one built-in multiplexer, and its relatively high speed\(^\text{16}\). A picture of a portion of the prototype for this section appears in Figure 15.
A set of 8 connectors on this prototype are designed to interface to the eight channel analog prototype. Their pinout appears in Table 3 and corresponds to that of the analog section when using ribbon cable connectors. The prototype is connected to the digital section by two identical connections. Each connector carries the control and data lines for four analog to digital converters as well as power for the entire board. The pinout for each type of connector appear in Table 3 and Table 4.

Table 3: Pinout for input from amplifier board to ADC prototype.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdd</td>
<td>Analog in</td>
</tr>
<tr>
<td>Vdd</td>
<td>Analog in</td>
</tr>
<tr>
<td>Reference</td>
<td>Analog in</td>
</tr>
<tr>
<td>Reference</td>
<td>Analog in</td>
</tr>
<tr>
<td>Ground</td>
<td>Analog in</td>
</tr>
<tr>
<td>Ground</td>
<td>Analog in</td>
</tr>
<tr>
<td>Ground</td>
<td>Analog in</td>
</tr>
</tbody>
</table>

Figure 15: Portion of the analog to digital prototype. (Robert Dyer)
The ADC board includes the reference supply for the ADC reference voltage as well as the amplifier circuitry. The ADC prototype also includes the voltage regulators for the ADC section and the amplifier section. They are described in more detail in the power control section. The interface to the MAX1281 uses the serial peripheral interface (SPI), a synchronous serial protocol that requires 3 lines: Sclk, Data In, and Data Out.

**Wireless System Design**

The wireless link is the TEAS module’s only communication link with the outside world other than the inductive charging link. The radio link must be powerful enough to reach a receiver in the room containing the subject. Three to five meters is an acceptable minimum for laboratory testing. A prosthesis on the subject’s body would require even less range. Any range calculation must take into account the fact that the antenna is under the subject’s skin and thus somewhat shielded. The link should be resistant to interference and should not cause excessive noise in other parts of the circuit. Ideally, several such links should be able to coexist within range of each other. As described in the design overview, the minimum baud rate for real-time transmission of 64 channels of spike timing data is around 300 kbits/s. Transmission of non-real-time (i.e. buffered) spike shape requires around 450 kbits/s. These minimums do not take into account the bandwidth requirements of communication protocol overhead. This overhead includes error correction, synchronization, packet splitting, etc. As with all the electronics in the
module, low-power consumption and small size are essential characteristics of the wireless system.

**Possible Systems**

There are a variety of commercial wireless systems that might be suitable for the TEAS radio system. A brief overview of the available technologies is helpful in understanding the trade-offs involved in selecting a wireless system.

**802.11b**

802.11b is a popular protocol for wireless personal computer networking. 802.11b radios typically have a range between 10 and a few hundred meters. The standard data-rate for this protocol is 11 Mbits/s. The protocol uses spread-spectrum techniques to increase the robustness of the link and allow several transmitters operate in the same physical area. Typical 802.11 transceivers are packaged PC cards.

**Cell phones**

One of the first ideas proposed to implement the TEAS wireless link was the idea of using a cellular phone. Several protocols exist for wireless data transmission over cellular telephones. Though new cell phones are supposed to provide high bandwidth, all of the available protocols are relatively low bandwidth and are typically geared at web surfing, though some also allow modem connections. In any case, the interface to these devices is complicated. The range of these protocols is far greater than what is necessary for the TEAS project. This excess range translates into excessive power requirements. Even if a cell-phone were stripped down to the minimal circuitry needed for wireless data transfers, it would be physically too large, too power hungry, and too difficult to use.

**FSK module**

Frequency shift keying (FSK) modules are often used to implement wireless links in experimental apparatus. A typical example of such a module is the AT86RF211 from Atmel. This module is a FSK modulated transceiver designed for use in the license-free
industrial, scientific, medical (ISM) frequency bands between 400 and 950 MHz. The AT86RF211 allows data links at up to 50 kbits/s. These modules consume relatively little power (e.g. 30 mA at 3 V during transmit). Typically, FSK modules are designed for a single frequency and thus are vulnerable to point sources of RF noise and interference between modules; however, some modules (such as the Atmel part) allow the user to implement frequency hopping spread spectrum (FHSS) schemes to circumvent both of these problems; however, designing such a protocol is fairly complicated. The interface to an FSK module is typically quite simple, usually some sort of serial data protocol with minimal control signals.

**Spread-Spectrum Radio Module**

Several companies produce spread-spectrum transceiver modules. These modules are usually designed to be controlled by a microcontroller or a baseband controller for a wireless protocol. Atmel, Motorola, National Semiconductor and other companies offer radio modules whose transmission frequency or modulation parameters can be controlled externally. This control, though it gives the designer a lot of flexibility, requires the radio link to be designed at the hardware, software, and protocol level. In order to build a robust wireless link as rapidly as possible, it is preferable to choose a system that is more constrained but easier to design. Commercial Bluetooth modules are one example of such a system.

**Bluetooth**

Bluetooth is a royalty-free wireless standard, originally designed to seamlessly replace wires in applications such as cell-phone headsets, portable computer "hot-syncing," and workstation wiring (e.g. printer, monitor, and mouse cables). As such, the Bluetooth protocol was specifically designed for short range, low power, relatively high bandwidth communication. The current Bluetooth standard specifies a data transfer rate of 725 kbit/s. As discussed earlier, this transfer rate is sufficient for the current needs of the TEAS project. Since it is designed to easily replace existing wired links, the Bluetooth hardware is designed to have relatively simple interfaces. Typical controllers are controlled with simple serial commands or USB connections.
**Bluetooth Design**

A Bluetooth module seemed to offer the best combination of low-power, bandwidth, and ease of use. In order to understand the design of the TEAS Bluetooth design, it is helpful to understand the basic Bluetooth protocol stack. The Bluetooth standard defines several protocol layers, some of which are shown in Figure 16.

![Bluetooth Protocol Stack Diagram](image)

**Figure 16: Lower part of the Bluetooth protocol stack.**

The baseband protocol layer defines the interface between logical signals and the physical modulation of these signals. The baseband is controlled by the one of two data-link protocol layers: the link control manager (LCM) controls the radio-to-radio communication necessary for setting up a link, synching clocks, and all the "housekeeping" necessary to maintain the link. The logical link control and adaptation layer protocol, referred to as L2CAP, provides connection creation and maintenance services to higher communication layers. Thus, the LCM layer handles the logic necessary to maintaining a link and the L2CAP provides the interface for passing one or more datastreams through such a link. The basic protocol layer that can be controlled by an external, non-Bluetooth device is the host controller interface (HCI). The HCI is a relatively simple command based protocol that offers commands for detecting Bluetooth devices in communications range, creating links to these devices, and transmitting and
receiving data packets over these links. The Bluetooth protocol defines many higher-level protocols: RFCOMM (a serial link replacement protocol), service discovery protocol (SDP), and many others. However, at the moment, no commercial Bluetooth module implements these higher-level protocols; they are left to the external controller.

Because the higher-level Bluetooth levels currently need to be implemented in software we chose to use the Bluetooth specification up to the HCI level. This level is available in several commercial controllers and combines ease of use with low-level flexibility.

Selection of Bluetooth Module

In late 2000, Bluetooth was still a young technology. Very few commercial products existed and many manufacturers of Bluetooth devices were still developing their chipsets. Availability would prove to be one of the primary criteria in selecting a module. From the technical standpoint, the Bluetooth module used in TEAS needs to implement the Bluetooth protocol at least up to the HCI level. Higher protocol layers such as RFCOMM might be convenient but are not essential. Like all the electronics in the system, the Bluetooth circuitry must be small and low power. Though Bluetooth devices are designed for space-constrained applications, most available designs use two or more integrated circuits and dozens of passive components. Typical designs have an RF chip and a baseband controller that implements some portion of the bottom of the Bluetooth protocol stack. Though we looked at some devices from manufacturers who built only one of the two (e.g. National Semiconductor’s RF chip, Brightcom’s IntelliBLUE™, and Broadcom’s BCM2039 baseband controllers), the difficulties involved in interfacing chips from different manufacturers seemed overwhelming. The problems associated with assembling fine-pitch ball-grid-array (BGA) packages and obtaining RF passive components (especially inductors) also militated against integrating chips from one or more manufacturers. Thus, we focused our search on Bluetooth modules: small printed circuit boards (PCBs) or multichip modules containing all of the necessary hardware to implement a complete Bluetooth link. At the beginning of 2001, only a few manufacturers that had announced such modules. The most prominent included Ericsson™ (one of the original designers of the Bluetooth protocol), Cambridge Silicon
Radio\textsuperscript{31} (CSR), Silicon Wave\textsuperscript{32}, Atmel\textsuperscript{33}, and Philips\textsuperscript{34}. The latter two seemed to have interesting products but were unable to promise availability of working devices before late 2001. Silicon Wave has designed a two-chip module entirely based on CMOS technology, which thus promises low power and low cost; however, manufacturing delays and their extreme unwillingness to sell small quantities of their devices removed them from contention. The CSR and Ericsson modules are very comparable in terms of functionality. Both modules implement the Bluetooth protocol stack through the HCI level (both companies claim their chips will do more than that soon). Both modules are comparable in size, though the CSR PCB is a slightly smaller that Ericsson’s. The two devices consume approximately 50 mA during transmit. They can both be controlled via USB or serial interfaces (UART in the case of the CSR device; I\textsuperscript{2}C for the Ericsson). However, the Ericsson module is limited to 460 kbits/s on its serial link while the CSR module’s UART is able to achieve 1.5 Mbits/s\textsuperscript{21}. The higher speed interface is essential if we are to achieve the maximum throughput of the radio link. In addition, CSR has a concrete plan for reducing the size, cost, and power consumption of their module over the next few years. If they are successful, the TEAS project will be able to benefit from these improvements without significant changes to the radio interface. Finally, another group at MIT had acquired the CSR modules and thus made it possible to build the TEAS communication link without waiting for CSR.

\textbf{CSR Module}

The CSR module (actually assembled by Mitsumi) measures $23 \times 13.6 \times 2$ mm with 30 surface mount, side-contact, connections. A photo of one of the modules used in the project appears in Figure 17.
Figure 17: Actual size Bluetooth module prior to assembly. (Robert Dyer)

The module has three interfaces that are relevant to the TEAS project: USB, UART, and SPI. The USB interface will be interesting if we ever want to make a receiver station that can be easily used with many different computers and operating systems: it is irrelevant for the surgically imbedded module. The UART interface is the most generally useful; it allows data transfer and control over low-level properties of the module. A few low-level properties such as interface control are most easily controlled with the SPI interface. SPI is a synchronous serial protocol often used by microcontrollers. Our approach to controlling the modules has been to use the software provided by CSR to set the interface parameters with a PC and the CSR SPI link. All other control is done via UART. The CSR module’s general specifications appear in Table 5.

Table 5: Basic characteristics of the CSR Bluetooth module.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vdd</td>
<td>2.7 to 3.3 V</td>
</tr>
<tr>
<td>Current (receive mode)</td>
<td>41 mA</td>
</tr>
<tr>
<td>Current (transmit mode 725 kbits/s)</td>
<td>41 mA</td>
</tr>
<tr>
<td>Current (scan mode)</td>
<td>560 μA</td>
</tr>
<tr>
<td>Park (low power quiescent)</td>
<td>120 μA</td>
</tr>
<tr>
<td>Range (without amplifier)</td>
<td>10 m</td>
</tr>
</tbody>
</table>

The module’s park mode allows it to “sleep” while occasionally checking for radio activity. This mode will allow the TEAS module to save power when it isn’t being used
as the radio and microcontroller will always be on. The ten-meter range specified above is for the standard, low-power CSR module with a PCB antenna and no power amplifier. Ten meters should be sufficient; however, if more power is needed in order to penetrate the subject’s skin or for some other reason, the external radio could use one of the higher power modules that CSR claims have ranges of several hundred meters.

The CSR module pinout appears in Table 6. In the final design only the UART receive (UART_RX) and transmit (UART_TX), the antenna (ANT), and the power (GND and VDD) connections are necessary.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PIO[0]/RX_EN</td>
<td>GND</td>
</tr>
<tr>
<td>2 PIO[1]/TX_EN</td>
<td>SPI_CSBI</td>
</tr>
<tr>
<td>3 GND</td>
<td>SPI_MOSI</td>
</tr>
<tr>
<td>4 GND</td>
<td>SPI_CLK</td>
</tr>
<tr>
<td>5 PIO[4]/IRQ1</td>
<td>SPI_MSO</td>
</tr>
<tr>
<td>6 PIO[5]/IRQ2</td>
<td>UART_CTS/USB_D-</td>
</tr>
<tr>
<td>7 PIO[6]</td>
<td>UART_RTS/USB_D+</td>
</tr>
<tr>
<td>8 PIO[7]</td>
<td>UART_RX</td>
</tr>
<tr>
<td>9 PCM_OUT</td>
<td>UART_TX</td>
</tr>
<tr>
<td>10 PCM_CLK</td>
<td>PIO[3]</td>
</tr>
<tr>
<td>11 PCM_IN</td>
<td>PIO[2]</td>
</tr>
<tr>
<td>12 PCM_SYNC</td>
<td>N/C</td>
</tr>
<tr>
<td>13 GND</td>
<td>GND</td>
</tr>
<tr>
<td>14 VDD</td>
<td>ANT</td>
</tr>
</tbody>
</table>

The maximum range of the modules is only obtainable with a good external antenna. However, 10 meters is quite possible with a PCB trace cut to the suitable dimensions (e.g. ¼ wavelength: 31 mm). Even a short wire gives adequate performance. We obtained
full-speed transmission over several meters with wire stubs cut to approximately 31 mm length.

Testing and Implementation

Though the CSR modules are fairly well documented and are fairly straightforward to work with from the hardware perspective, they must be controlled carefully if one wants to transmit data as fast as possible. Our initial tests used the CSR Casira development kit\textsuperscript{31}. This kit consists of two discs; each disk contains a Bluetooth transceiver and interface circuitry to connect the disk to a PC using RS-232, USB, and SPI. CSR provides a C++ program that allows text messaging and file transfer between the two disks. This software is not particularly useful as a model for the TEAS Bluetooth control system since it isn’t optimized for speed or simplicity. However, by examining the program’s serial output and comparing it to the Bluetooth specification, it was possible to begin to create our own Bluetooth software.

The HCI Interface

The first step in controlling the CSR modules is to reprogram them to accept HCI (which CSR sometimes call the H4 transport) commands at the desired baud rate. CSR ships the modules programmed to operate with their BlueCore\textsuperscript{tm} Serial Protocol (BCSP). This protocol allows several levels of the Bluetooth protocol stack to be transmitted easily over the same UART link in logically separate tunnels. BCSP is not useful to TEAS since we will only use the HCI level and we want to minimize communication overhead. The UART protocol as well as many other properties of the module can be changed with the PSTools software from CSR. The newer versions of this software allow properties to be changed over any of the three data interfaces (UART, SPI, USB); however, the underlying interface for this is SPI. It is possible to lock oneself out of the modules by misprogramming some of the variables available in PSTools. It should always be possible to recover from such a mistake by using the SPI protocol. It is also possible to erase unrecoverable data from within this program; in particular, the power table and oscillator settings are critical to the operation of the module, vary from module to
module, and can be irretrievably corrupted. Table 7 shows the variables that need to be set in order to program the modules for HCI control over a UART using 1 stop bit, and no parity\textsuperscript{22}.

Table 7: Settings necessary to reprogram the Bluetooth module for HCI mode

<table>
<thead>
<tr>
<th>Host Control Interface</th>
<th>H4 UART</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop Bits</td>
<td>1</td>
</tr>
<tr>
<td>Parity</td>
<td>None</td>
</tr>
<tr>
<td>Hardware Flow Control</td>
<td>Enable</td>
</tr>
<tr>
<td>RTS asserted</td>
<td>Enable</td>
</tr>
<tr>
<td>BCSP Hardware enable</td>
<td>Enable</td>
</tr>
</tbody>
</table>

When the module is correctly configured for 115 kbits/s, operation the UART configuration bitfield should be 168. Some versions of PSTools do not give the user access to all of these fields. In this case, hitting Ctrl-Alt-a will switch the program into "super-user" mode and give one access to all fields.

UART Packet Structure

CSR uses a modified version of the Bluetooth UART/RS-232 specification. Packets sent over the UART link must be preceded by an 8-bit packet indicator. The indicators are listed in Table 8.
Table 8: UART packet indicators used in the Bluetooth standard.

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Packet Indicator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Packet</td>
<td>0x01</td>
<td>HCI command</td>
</tr>
<tr>
<td>ACL Data Packet</td>
<td>0x02</td>
<td>Data packet</td>
</tr>
<tr>
<td>SCO Data Packet</td>
<td>0x03</td>
<td>Encoded sound data</td>
</tr>
<tr>
<td>Event Packet</td>
<td>0x04</td>
<td>Module response</td>
</tr>
<tr>
<td>Error Message Packet</td>
<td>0x05</td>
<td>UART connection control</td>
</tr>
<tr>
<td>Negotiation Packet</td>
<td>0x06</td>
<td>and negotiation (not used)</td>
</tr>
</tbody>
</table>

The first four indicators precede HCI information. The last two are to be used to control and negotiate UART connections. Not all CSR modules implement this negotiation capability, and the TEAS software doesn’t use them. According to the Bluetooth specification, the packet indicator should precede an eight-bit sequence number; however, the CSR modules do not seem to implement this part of the specification. The rest of a UART packet is simply an HCI packet. HCI commands are preceded by the command packet indicator; events (typically responses generated by the module) are preceded by the event packet indicator. SCO data packets represent encoded sound data and thus are not used in TEAS.

**HCI Packet Structure**

Table 9 shows the basic structure of an HCI command packet. The opcode is divided into two parts: the OpCode Group Field (OGF) and the OpCode Command Field (OCF). The 6-bit OGF is shared between families of commands. The 10-bit OCF specifies the command\(^{22}\).
Table 9: Structure of a Bluetooth HCI command packet.

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>OCF-command</td>
<td>OGF-group</td>
<td>Parameter Total Length</td>
<td>Parameter 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter 1</td>
<td>Parameter 2</td>
<td>Parameter 3</td>
<td>Parameter 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter N-1</td>
<td>Parameter N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10 shows the format for an HCI event packet. Event packets are sent by the CSR module to acknowledge commands, respond to queries, or signal error conditions. Each event code specifies the type of event and the parameters contain the various parts of events.

Table 10: Structure of an Bluetooth HCI event packet.

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Code</td>
<td>Parameter Total Length</td>
<td>Parameter 0</td>
<td></td>
</tr>
<tr>
<td>Parameter 1</td>
<td>Parameter 2</td>
<td>Parameter 3</td>
<td></td>
</tr>
<tr>
<td>Parameter N-1</td>
<td>Parameter N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11 shows the structure of an HCI Data packet. An HCI data packet can flow either from the controller to the Bluetooth module (when the controller is transmitting data) or in the opposite direction when the module receives data over the Bluetooth link. The first 12 bits are the connection handle. In a multi-connection environment, this handle would indicate which Bluetooth connection produced the data or was to receive it. The TEAS system only uses one connection handle, which is set when the Bluetooth connection is initiated. The PB Flag is two bit flag: it should be set to \(0b10\), the indicator for the first
packet of a higher-level message. The next two bits set the broadcast mode; \texttt{0b00} will set the broadcast mode to point-to-point.

Table 11: Structure of a Bluetooth HCI data packet.

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>16</td>
<td>24</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB</td>
<td>BC</td>
<td>Data Total Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data

**Controlling the CSR Module**

Appendix A contains the complete details on the commands used to control the modules and set up a communication link. This section will only give the general sequence of interactions without bit-level details. The sequence described is the one used by our first Java based test program, BlueGUI. This program allowed a user to set up a connection and transmit data between two personal computers. The final version also demonstrated automatic behavior by responding to certain commands received over the Bluetooth link: in particular it was possible to test the divisibility of a number sent with this test command. Figure 18 shows a screen shot of this program and Appendix B contains the complete source code.
A Bluetooth connection links two devices. One of these devices is designated the master and initiates the link with the other (slave) module. Table 12 shows the sequence of commands and responses for a master module.
Table 12: Command sequence for setting Bluetooth master mode.

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Command sent to master module</th>
<th>Master module response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Command status event</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Command complete event</td>
</tr>
<tr>
<td>4</td>
<td>Inquire (ping for devices in range)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Command status event</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Inquiry result (slave device detected)</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Command complete event</td>
</tr>
<tr>
<td>8</td>
<td>Connect</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Command status event</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Connection complete</td>
</tr>
<tr>
<td>T+1</td>
<td>ACL Data Packet</td>
<td>Number of completed packets event</td>
</tr>
<tr>
<td>T'</td>
<td></td>
<td>ACL Data Packet (incoming)</td>
</tr>
</tbody>
</table>

Most of the commands are replied to by command status events and command complete events. These are mainly useful in verifying that the command was received and correctly parsed. If the command completes fast enough, the command status event will not be generated. The inquiry result event contains the Bluetooth device address of the slave module detected along with other parameters necessary for creating the connection. The connect command contains a field for clock offset. There are Bluetooth commands for discovering this parameter; however, this parameter is not necessary in the context of a single point-to-point connection. The connection complete event contains the channel number that must be used to generate the ACL data packet.

Setting up a module for slave mode is a little bit more complicated since the module must be programmed to scan for incoming inquiries and connection requests. The frequency of this scan determines the speed of response of the unit as well as the amount of energy consumed by the module while it waits: the module is in its low-power park mode between scans. It is important to match the length of time the master sends inquiries to the frequency of scan. If the inquire sequence is too short, it is possible for the master to miss the slave. Table 13 shows the sequence of commands that puts a module into slave mode.
Table 13: Command sequence for Bluetooth slave mode.

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Command to slave module</th>
<th>Slave module response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Command status event</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Command complete event</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Set event filter</td>
<td>Command complete event</td>
</tr>
<tr>
<td>5</td>
<td>Command complete event</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Write scan enable</td>
<td>Command complete event</td>
</tr>
<tr>
<td>7</td>
<td>Command complete event</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>Connection complete event</td>
<td></td>
</tr>
<tr>
<td>T'+1</td>
<td>ACL Data Packet</td>
<td>Number of completed packets event</td>
</tr>
<tr>
<td>T''</td>
<td>ACL Data Packet (incoming)</td>
<td></td>
</tr>
</tbody>
</table>

The set_event_filter sets the module to accept connections automatically while the write_scan_enable command instructs the module to listen for inquiries on a schedule determined by the write_inquiry_scan_activity command. The exact bit sequences needed for these commands are in Appendix A. The connection complete event is generated when the master module creates a connection.

In order to maximize the throughput of the Bluetooth system, it is necessary to use a system of tokens to take advantage of the module’s packet buffer and reduce communications overhead. The CSR modules can buffer up to eight data packets. Thus, the controller can send up to eight packets without waiting for a response from the module. The module can acknowledge the transmission of these packets in groups, thus minimizing the return traffic to the controller. This reduction in overhead is particularly significant in the case of the controller implemented by the MSP430 microcontroller that needs to spend the majority of its time performing other tasks.

The MSP430 Bluetooth control program is very similar to the Java test program described above except that all functions are under automatic control. A stripped-down version appears in Appendix C. This version of the program only demonstrates simple Bluetooth communication and does not demonstrate complicated error correction. A
similar program was used to transmit high-speed data for a test in another lab project. That program collected data from the MSP430’s analog to digital converter at 5 kHz and transmitted the data over a Bluetooth link to a laptop. The user can control the data acquisition module by sending simple, 1-byte commands in the opposite direction. A picture of the MSP430 development kit connected to a Bluetooth module for this test appears in Figure 19. Notice the short, red wire stub on the right that serves as an antenna.

![Figure 19: Bluetooth module, wire antenna, and MSP430 connected for a test. (Robert Dyer)](image)

**Power System Design**

The TEAS power system is under control of the microcontroller. In the lowest power mode the microcontroller, the Bluetooth module, and their regulator are the only chips receiving power. The other devices in the system are controlled by separate regulators. The other power sections are: amplifier section, analog-to-digital converter, and digital section (FPGA and SRAM). Each regulator is one of two in a LM2966 chip, a low-dropout, 3 V, dual regulator. Each channel of the LM2966 has an independent shutdown circuit that allows simple control of the power system. These separate regulators also serve to provide a bit more power-noise isolation between the parts of the implant.
The microcontroller is also responsible for managing battery charging. Its analog to
digital converter can compare the battery voltage to an internal reference. When
appropriate, the microcontroller signals the need for charging to the outside world and
enters low-power mode. The microcontroller is not required for charging; thus the
system can be recharged even if the batteries are so drained that microcontroller ceases
operation. However, if the microcontroller is operational, it supervises the charging
process, detecting overheating and other error signals sent by the charging circuitry.

**Testing**

**Amplifier**

Initial tests of the analog system focused on amplifying a 100 µV signal in the presence
of noise. The simplest way to accomplish this was to amplify signals from a resistor-
divider fed by a pattern generator. Figure 20 shows the schematic of a test circuit built
into the first amplifier board.

![Schematic of test circuit](image)

**Figure 20:** 100 µV signal generators with and without impedance control. (Jan Malášek)

This circuit was also useful for verifying the frequency performance of the amplifier.
The final version of the amplifier amplifies signals without problems even with long (100
mm) unshielded input wires in our lab’s fairly electrically noisy environment.
Input impedance is the other significant effect on the amplifier board. We attempted to simulate the impedance of the neuron-electrode connection with an RC parallel circuit tuned for 2 MΩ impedance at DC and between 100 and 500 kΩ at 10 kHz. These impedance values are believed to mirror those produced by current electrode arrays. Figure 20 also shows the schematic of this test circuit. These tests were less conclusive. Johnson noise and noise due to the amplifiers' bias current introduced significant noise into the output.

A true test of the amplifiers' general performance, and in particular its behavior when attached to an electrode array, requires in vivo testing. Our first attempt to make such a test was also the initial test of our group's electrode array. Unfortunately, this test produced no useful data. The electrode did not produce any useful signals even when attached to our collaborators' current electronics system. Further testing using our array has not yet been possible. Interface and time constraints have made other forms of in vivo testing impractical.

**Bluetooth**

Speed and signal quality testing of the Bluetooth system was performed by transmitting known sequences between a Bluetooth module under PC control and one under microcontroller control. The PC was equipped with a ConnectTech BlueHeat fast serial card to allow the PC to communicate at 1.3 Mbits/s (1382400 bits/s). The current maximum speed of the Bluetooth transfer is 490 kbits/s. The distance between the transmitters in these tests was two meters: increasing range will degrade the performance but not before approaching the 10 meter maximum range. This throughput compares favorably with the numbers obtainable with CSR's software, even though the theoretical maximum throughput of the module is 725 kbits/s. The speed doesn't seem to be limited by the Bluetooth devices but rather the PC used at one end of the link. 490 kbits/s was also the maximum throughput obtained when the microcontroller was directly connected to the PC. Improving the serial driver or changing the operating system of the PC should improve the throughput. However, even 490 kbits/s is more than sufficient for the initial testing TEAS implant.
**Conclusions, Future Work**

**Final Prototype Specification**

Though we have not yet built an implantable prototype of the TEAS device, we already know many of its characteristics from our current prototypes. Probably the most important characteristics of this implant are its size and power consumption. The size of the implant is constrained by our desire to implant it subcutaneously on the cranium of one of the non-human primates trained by our colleagues at Brown University. The power requirements are constrained by the need to minimize recharging time and the need to avoid overheating the tissue around the implant.

**Area Requirements**

Most of the subjects used at our collaborators lab are macaque monkeys. A plastic cast of a *Macaca mulatta* skull appears in Figure 21. Some of the space on the top of the cranium must be conserved for the electrode and the trans-cranial opening. The available space on the back of the cranium is approximately 5 mm on a side.
Table 14 provides a rough estimate of the area occupied by the chips in each of the sub-units of the implant. The area needed for interconnect depends on the number of levels used. Six layers are a minimum if the implant is to have adequate ground and power planes as well as adequate interconnect. Six-layers also allow us to make a fairly flexible PCB that will conform to the cranium and allow us to make the best use of the space available to us.
Table 14: Area estimate for subunits of the final prototype.

<table>
<thead>
<tr>
<th>Subcircuit</th>
<th>Chip Area (mm)$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog</td>
<td>946</td>
</tr>
<tr>
<td>ADC</td>
<td>400</td>
</tr>
<tr>
<td>Microcontroller</td>
<td>170</td>
</tr>
<tr>
<td>FPGA/SRAM</td>
<td>610</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>350</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>400</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2876</strong></td>
</tr>
</tbody>
</table>

The chip area in Table 14 includes the area needed to place and solder the chip but does not account for other interconnect. Tripling the chip area gives a reasonable estimate of the total layout area. It will be desirable to reduce the area of the prototype by separating the implant into two stacked circuit boards. A reasonable place to make this separation is between the analog to digital converters and the FPGA. This separation exists in our current prototypes and is relatively easy to make: only 23 connections need to be made between the two sides of this split. 23 connections can easily be accommodated with a small pitch PCB-to-PCB connector. Stacked circuit boards would not conform as well to the cranium but since the overall area is smaller and the increased thickness is acceptable from a surgical perspective, it is probably preferable. In addition, the analog section could probably be manufactured on a cheaper, more flexible four-layer PCB. Separating the analog amplifiers and the main digital section will also reduce noise problems in the analog section. Even quadrupling the chip area above (a conservative estimate of the overall layout area) indicates that two double-sided boards 50 mm on a side should be sufficient to accommodate the layout.

**Power Requirements**

The chart below gives an approximate idea of the power requirements of the implant (Table 15).
Table 15: Power consumption of implant subunits.

<table>
<thead>
<tr>
<th>Subunit</th>
<th>Power Consumption (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth module</td>
<td>180</td>
</tr>
<tr>
<td>Microcontroller</td>
<td>9</td>
</tr>
<tr>
<td>Amplifiers</td>
<td>360</td>
</tr>
<tr>
<td>ADC</td>
<td>60</td>
</tr>
<tr>
<td>FPGA/SRAM</td>
<td>450</td>
</tr>
<tr>
<td>Power management</td>
<td>30</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1089</strong></td>
</tr>
</tbody>
</table>

The overall current requirement of the system is under 400 mA, which is well within the typical maximum discharge rate of a lithium polymer battery. Since the onboard electronics all work at 3 V, a lithium-polymer battery with a low-dropout linear regulator was chosen (the nominal voltage of the lithium-polymer battery is 3.6 V). A typical Lithium-polymer battery used in some cell-phones is rated for 15,552 J (1.2 Ah). At a discharge rate of 400 mA and derating the capacity to account for the fact that the battery voltage must stay over the 3 V supply voltage, a 30 gram battery 50 mm on a side and a few millimeters thick will provide 2 or 3 hours of power. Since Lithium-polymer can be made somewhat flexible, it may be possible to place such a battery on the cranium. However, for testing in monkeys, it is preferable to implant the battery in the chest or peritoneal cavity. This placement facilitates charging and allows for a slightly larger battery with increased capacity.

**Encapsulation**

The chemical environment between the cranium and the skin is slightly less harsh than that found in the brain itself; however, it is still essential to protect the implant electronics from corrosive bodily fluids. Obviously, it is also necessary to protect the subject from contact with non-biocompatible materials that could cause infection or toxic reactions. The implant housing should also be relatively smooth and conform to the skull in order to make surgical implantation as simple as possible. A titanium shell is one possibility. Titanium has the advantage of being very biocompatible and solid enough to protect the
implant from any likely physical shock. A conductive shell such as this would also make for a convenient ground reference. However, a titanium shell would not easily fit itself to the subject’s skull, is relatively difficult to manufacture, and will tend to act as a faraday cage, which might put severe constraints on the Bluetooth link. Another possibility is to coat the populated flexible-PCB with a thick coat of parylene\(^5\). Parylene is the biocompatible coating material used to protect the electrode array assembly. It is flexible and would not interfere with radio transmission. Of course a parylene coat would not provide much physical protection and would only be of marginal usefulness in smoothing sharp corners. These problems could be resolved by first physically protecting the implant inside a rubber pocket and then coating the pocket with parylene so as to make the entire module more biocompatible.

**Current status**

Most of the subsystems for the TEAS have been tested individually and when integrated with the other systems. The analog front-end should be tested *in vivo*, preferably, but not necessarily, with our group’s array. The only sub-system that is not ready to be integrated into a final implant is the contactless charging system. However, this subsystem is separate from the main implant and thus doesn’t limit initial testing of a real-size implant. Another member of our group (Fedor Danilenko) has demonstrated contactless charging. Unfortunately his system is not ready for use with live subjects: it provides too little current, requires high voltage power supplies, and does not address the problem of attaching the charging equipment (probably part of a belt or jacket) on an uncooperative subject.

**The Next Step**

The largest task facing the TEAS electronics team is that of developing adequate software for use with the system. Our collaborators at Brown University currently use a system from Plexon Inc. to acquire and analyze neural signals. Our end-user interface must provide similar functionality: graphing, graphical trigger setting, real-time displays, overlays of successive potential neural spikes, as well as basic storage and interface capability. Raw data dumps are completely inadequate in an environment with
uncooperative subjects, electrical noise, and data that is fundamentally hard to analyze. Even testing of something as simple as a single analog channel is difficult in such an adverse environment.

Laying out and manufacturing a test implant will be relatively straightforward. The existing prototype layouts can be used as a starting point. They are all two-layer boards; changing the layouts to use ground and power planes should simplify this layout. Many of the prototype layouts have tightly packed layouts around each chip but widely separated; reducing the interchip spacing should take care of most of the layout problems for these circuits. Manufacturing will be greatly simplified by the availability of all the parts.

The current Bluetooth link is acceptable for a first implant. Improving the external (computer) part of the link with a faster interface (faster RS-232 or USB), though desirable, is not a priority. Replacing the current module with a new module could reduce the power consumption and the size of the Bluetooth section. CSR, the manufacturer of our current module, has announced one such module; however, acquiring it has so far proven impossible. This module would maintain our current interface but cut the power consumption and size of the module in half.

As described elsewhere\textsuperscript{12}, the first test of the array was not successful. However, all the electronics-related components (flex circuit board, connector) seem to have worked as designed. Thus, for the moment, no redesign of the electronics implant is necessary.

**The Future**

At this stage of the design process COTS components are not a viable choice for anything but the very first implant. Our parts are cutting edge but are still too large and not precisely adapted to our design goals. A first step in improving the implant would be to design a single chip amplifier circuit for all the amplifiers in the array. Such a chip might integrate some second stage multiplexing as well as improved filtering. Similar VLSI approaches to the analog front-end have been proposed and even designed by other
researchers\textsuperscript{8}. If the resulting die could be made small enough, it may even be possible to mount the die on the back of the array, thus minimizing the noise introduced by the array-electronics interconnect. Integrating the ADCs and the digital section is not as necessary. These components are not as large and are highly optimized for exactly the purpose they are being used for. It would probably be possible to replace the FPGA by a lower power, and smaller ASIC; however, this improvement would come at the cost of flexibility. Some improvements in size could be made by replacing the digital chips by bare-die. Unfortunately, bare-die introduce additional manufacturing difficulties and are very difficult to acquire.

While we were working on the implant, our colleagues at Brown University have continued to improve their understanding of neural encoding of muscle motion. In particular, they have demonstrated decoding of limited arm motion via their current non-wireless system\textsuperscript{10}. The motion decoded was actually that of a computer cursor under the control of a pen tablet. This decoding is less than full 3D control but is a large step in the direction of wireless brain control of prostheses. A similar demonstration using a wireless brain-machine interface should be possible in the near future.
References:

Appendix A: Bluetooth Bit-level Commands

The following table presents the HCI commands essential for setting up and using a Bluetooth data link for the device performing as the Bluetooth master. The hexadecimal commands are in standard PC bit ordering.

<table>
<thead>
<tr>
<th>Bluetooth Command</th>
<th>Hexadecimal Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0x01030C00</td>
</tr>
<tr>
<td>Inquire</td>
<td>0x01010405338B9E0303</td>
</tr>
<tr>
<td>Connect</td>
<td>0x0105040D + bdaddr + &quot;cc180100&quot; + clkOffset + &quot;00&quot;</td>
</tr>
<tr>
<td>ACL Data Packet</td>
<td>0x022620</td>
</tr>
</tbody>
</table>

For the Connect command clkOffset (the Clock Offset between the two devices) and bdaddr (the Bluetooth device address of the other device) can be derived from the inquiry result.

To setup a slave device the following additional commands are necessary.

<table>
<thead>
<tr>
<th>Bluetooth Command</th>
<th>Hexadecimal Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Event Filter</td>
<td>0x01050C03020002</td>
</tr>
<tr>
<td>Write Scan Enable</td>
<td>0x011A0C0103</td>
</tr>
</tbody>
</table>

Calculating the opcode of an HCI command packet can be confusing. The following algorithm will produce the appropriate bit sequence for standard PC bit order.

The OCF (OpCode Command Field, 10 bits) and the OGF (OpCode Group Field, 6 bits), form the opcode.

Write 000011;0000000001
OGF OCF
Cut at 8 bits 00001100;00000001
Swap 00000001;00001100
Insert as one-byte characters in the header or command.
Final OpCode: 0x010C

71
Appendix B: BlueGUI source code

This appendix contains the source code for the Bluetooth test program written in Java. The files included are:

- **Terminal.java**: The main control program opens a serial link to the Bluetooth module and controls all Bluetooth communication
- **SerialPortSetupException.java**: An exception used by Terminal.java
- **FatalException.java**: A terminal RunTime exception
- **BlueGUI.java**: The GUI for the test program: calls functions inside Terminal.java. BlueGUI was primarily written by Jan Malášek

**Terminal.java:**

```java
package bluetoothSerial;
import java.io.*;
import java.util.*;
import javax.comm.*;
import java.math.*;

public class Terminal {
    BufferedInputStream inStream;
    OutputStream outStream;
    //sport is only used as a handle for closing or such
    // use streams to read/write data
    SerialPort sport;
    //pg 685 of spec
    static String read_bd_addr = "01091000"; //READ BD_ADDR
    //from spec
    static String read_buffer_size = "01051000";
    //pg 542 of spec
    static String hci_inquiry = "01010405338b9e0303";
    //p. 602 of spec
    static String resetCommand = "01030c00";
    //pg 603 of spec
    static String auto_accept_connection_filter = "01050c03020002";
    //page 627 of spec
    static String write_scan_enable = "011a0c0103"; //inq and page scan
    on
    static String dataPacketHeader = "022620";

    public Terminal() {
    }

    public void start(String comName, int baudRate)
    throws SerialPortSetupException {
        /*
         * opens a port named comName at baudRate
         * if there is a problem throws SerialPortSetupException
         * sets up input output streams
         */
        //list known ports:
        printPortList();
        openPort(comName, baudRate);
    }
}
```

72
// setup flow-control Bluetooth doesn't seem to care
/* try {
   sport.setFlowControlMode(
   SerialPort.FLOWCONTROL_RTSCS_IN
   | SerialPort.FLOWCONTROL_RTSCS_OUT);
   //sport.setFlowControlMode(SerialPort.FLOWCONTROL_RTSCS_OUT);
   } catch (UnsupportedCommOperationException e) {
   throw new SerialPortSetupException("Comm port doesn't" +
   " support the flow " +
   "control parameters");
}*/

public void close() {
   sport.close();
}

public String readBdAddr() {
   send(hexStringToByte(read_bd_addr));
   return ("Read Bluetooth Device Address: " + read_bd_addr);
}

public String readBufferSize() {
   send(hexStringToByte(read_buffer_size));
   return ("Read buffer size: " + read_buffer_size);
}

public String inquire() {
   send(hexStringToByte(hci_inquiry));
   return ("Inquiry: " + hci_inquiry);
}

public String reset() {
   send(hexStringToByte(resetCommand));
   return ("RESET: " + resetCommand);
}

public String slaveMode() {
   send(hexStringToByte(auto_accept_connection_filter));
   send(hexStringToByte(write_scan_enable));
   return ("auto accept connect: " + auto_accept_connection_filter +
   "enable page and inquiry scan: " + write_scan_enable);
}

public String sendData(String sendThis) {
   return sendData(sendThis.getBytes());
}

public String sendData(byte[] sendThis) {
   int byteLength = sendThis.length;
   String reply = ""
   String dataPacket = dataPacketHeader;
   // according to my interpretation of read_buffer_size
   // the maximum data area for a packet is 128 bytes
   // length field is 2 bytes: LSB first
// turn the length field into hex
String lengthField = "0000";
if (byteLength < 16) {
    lengthField = "0" + Integer.toHexShort(byteLength) + "00";
} else if (byteLength < 128) {
    lengthField = Integer.toHexShort(byteLength) + "00";
} else if (byteLength == 128) {
    lengthField = "1000";
} else if (byteLength < 256) {
    reply += "String is too long. Here goes nothing\n";
    lengthField = Integer.toHexByte(byteLength) + "00";
} else {
    reply += "More than 255 bytes in data field...you're" +
    " crazy (I think) Ignoring this\n";
}

if (byteLength < 256) {
    dataPacket = dataPacket + lengthField;
    byte[] transmitBytes = glue(hexStringToByte(dataPacket),
sendThis);
    reply += "\>sending data: " +
    (new BigInteger(transmitBytes)).toString(16);
    // example data packet: (sends ascii 1 and 2)
    // dataPacket = "02262003310033000226200600020001003100"
    send(transmitBytes);
    // copied off bluechat session->actually 2 data packets
    String dataPacket = "02262006000200010033000226200600020001003100"
    }
    return reply;
}

public String connect(String bdaddr) {
    return connect(bdaddr, "0000")
}

public String connect(String bdaddr, String clkOffset) {
    String con_stub = "0105040d" + bdaddr + "cc180100" + clkOffset + 
    "00";
    send(hexStringToByte(con_stub));
    return ("creating connection: " + con_stub);
}

public String read() {
    String reply = "";
    byte[] tempData;
    tempData = receive();
    if (tempData.length>0) {
        reply += (new BigInteger(tempData)).toString(16);
        reply += "\n>" + (new String(tempData));
    }
    return reply;
}
public static void main(String[] args) {
    Terminal term = new Terminal();

    try {
        term.start("COM1", 115200);
        BufferedReader commandStream = new BufferedReader(
            new InputStreamReader(System.in));

        while (true) {
            //check RS232 for input
            System.out.println(term.read());

            String test = commandStream.readLine();
            if (test.equals("n")) {
                System.out.println("next");
            } else if (test.equals("slave")) {
                System.out.println(term.slaveMode());
            } else if (test.equals("reset")) {
                System.out.println(term.reset());
            } else if (test.equals("a")) {
                System.out.println(term.readBdAddr());
            } else if (test.equals("i")) {
                System.out.println(term.inquire());
            } else if (test.equals("b")) {
                System.out.println(term.readBufferSize());
            } else if (test.equals("d")) {
                System.out.print("Enter data to be sent: ");
                String dataString = commandStream.readLine();
                System.out.println(term.sendData(dataString));
            } else if (test.equals("c")) {
                //try to connect:
                String bdaddr = "250aff5b0200";
                System.out.println(term.connect(bdaddr));
            } else if (test.equals("")) {
                //do nothing
            } else {
                System.out.println("Unknown command: " + test);
            }
        }
    }
    catch(Exception e) {
        System.out.println("Caught: " + e + "\n" + e.getMessage());
        e.printStackTrace();
    }
}

private byte[] hexStringToByte(String input) {
    BigInteger bigint = new BigInteger(input, 16);
    byte[] byteVal = bigint.toByteArray();

    return byteVal;
}
private byte[] receive() {
    /* return an array of the current set of bytes from the input
     * stream
     */
    byte[] readBuffer = new byte[20];
    try {
        int numBytes = inStream.available();
        readBuffer = new byte[numBytes];
        inStream.read(readBuffer);
    } catch (IOException e) {
        throw new FatalException("Problem reading");
    }
    return readBuffer;
}

public String sendCommand(String hexdata) {
    // used to send a raw command from the UI
    send(hexStringToByte(hexdata));
    return ("Sent: " + hexdata);
}

private void send(byte[] data) {
    try {
        outStream.write(data);
    } catch (IOException e) {
        throw new FatalException("IO exception attempting to send
serial " +
        " data.");
    }
}

public void openPort(String spName, int baudrate) throws SerialPortSetupException {
    /* opens serial port and opens streams for inout and output
     * these streams are variables and should preserve
     * data across switches between read and write
     */
    // spName is the name of the serialPort we want (eg. COM1 in
windows)
    // appName is name of application which controls the serial port
    // if anyone else asks
    String appName = "JavaBluetoothSerialController";
    SerialPort serialPort = null;
    Enumeration portList;
    CommPortIdentifier portId;
    portList = CommPortIdentifier.getPortIdentifiers();
    while (portList.hasMoreElements()) {
        portId = (CommPortIdentifier) portList.nextElement();
        if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
            if (portId.getName().equals(spName)) {
                serialPort = (SerialPort) portId.openChannel();
                serialPort.setSerialPortParams(baudrate, 8,
PeerToPeerStream,protocolVersion,protocolVersion);
                serialPort.setBufferSize(CommPortIdentifier.PORT_BUFFER_SIZE);
if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL)
{
    if (portId.getName().equals(spName)) {
        //if (portId.getName().equals("/dev/term/a")) {
            try {
                serialPort = (SerialPort)
                    portId.open(appName, 2000);
            } catch (PortInUseException e) {
                throw new SerialPortSetupException(e.currentOwner +
                    " has" +
                    " the serial port");
            }
            try {
                serialPort.setSerialPortParams(baudrate,
                    SerialPort.DATABITS_8,
                    SerialPort.STOPBITS_1,
                    SerialPort.PARITY_NONE);
            } catch (UnsupportedCommOperationException e) {
                throw new SerialPortSetupException("Comm port
doesn't"+
                    " support the"+
                    " setup parameters");
            }
            /*try {
                serialPort.setFlowControlMode(
                    SerialPort.FLOWCONTROL_RTSCTS_IN
                    | SerialPort.FLOWCONTROL_RTSCTS_OUT);
            } catch (UnsupportedCommOperationException e) {
                throw new SerialPortSetupException("Comm port
doesn't"+
                    " support the flow "+
                    "control parameters");
            }*/
        }
    }
    if (serialPort == null) {
        throw new SerialPortSetupException(spName + " is not a valid"+
            " serial port.");
    }
}

//open input stream
try {
    InputStream tempStream;
    tempStream = serialPort.getInputStream();
    inStream = new BufferedInputStream(tempStream);
} catch (IOException e) {
    throw new FatalException("Problem opening inputStream");
}

//open output stream
try {
    outStream = serialPort.getOutputStream();
} catch (IOException e) {
    throw new FatalException("IO exception attempting to open serial" +
                        " port for writing.");
}

// set handle to SerialPort for closing and other such things
sport = serialPort;

public void printPortList() {
    /* prints list of known ports to STDOUT */
    SerialPort serialPort = null;
    Enumeration portList;
    CommPortIdentifier portId;

    portList = CommPortIdentifier.getPortIdentifiers();

    System.out.println("Known ports:");
    while (portList.hasMoreElements()) {
        portId = (CommPortIdentifier) portList.nextElement();
        System.out.println(portId.getName());
        //if (portId.getPortType() == 
        CommPortIdentifier.PORT_SERIAL) {
            //if (portId.getName().equals(spName)) {
        }
    }

    private byte[] glue(byte[] a, byte[] b) {
        /* returns an array which is a,b
         * either array can be of size zero
         */
        byte[] out = new byte[a.length+b.length];
        for (int i = 0; i < a.length; i++) {
            out[i] = a[i];
        }
        int offset = a.length;
        for (int i = 0; i < b.length; i++) {
            out[i+offset] = b[i];
        }
        return out;
    }
}

SerialPortSetupException.java:
package bluetoothSerial;

public class SerialPortSetupException extends Exception {
    SerialPortSetupException() {}
SerialPortSetupException(String msg) {
    super(msg);
}

FatalException.java:
package bluetoothSerial;

public class FatalException extends RuntimeException {
    FatalException() {}

    FatalException(String msg) {
        super(msg);
    }
}

BlueGui.java:
import bluetoothSerial.*;
import javax.swing.*;
import javax.swing.text.*;
import java.awt.*; //for layout managers
import java.awt.event.*; //for action and window events
import java.net.URL;
import java.io.IOException;

public class BlueGUI extends JFrame implements ActionListener {
    private String newline = "\n";
    protected static final String dataFieldString = "Data";
    protected static final String commandFieldString = "Command";

    protected JLabel actionLabel;
    protected static JTextPane textPane;
    protected JTextField addressField;
    protected static JTextField scaleField;
    protected static JTextField pollField;
    protected static JLabel resultLabel;
    private static String serialPortStatus = "Failed to open serial port\n";

    public static Terminal term;

    public BlueGUI() {
        super("BlueGUI");

        //Create a regular text field.
        JTextField dataField = new JTextField(10);
        dataField.setActionCommand(dataFieldString);
    }
dataField.addActionListener(this);
//Create a label for the fields.
JLabel dataFieldLabel = new JLabel(dataFieldString + " ":");
dataFieldLabel.setLabelFor(dataField);
//Create a label to put messages during an action event.
actionLabel = new JLabel("Type data and then Return to send");
actionLabel.setBorder(BorderFactory.createEmptyBorder(10,0,0,0));

//Create a regular text field.
JTextField commandField = new JTextField(10);
commandField.setActionCommand(commandFieldString);
commandField.addActionListener(this);
//Create a label for the fields.
JLabel commandFieldLabel = new JLabel(commandFieldString + " ":");
commandFieldLabel.setLabelFor(commandField);
//Create a label to put messages during an action event.
actionLabel = new JLabel("Type command and then Return to send");
actionLabel.setBorder(BorderFactory.createEmptyBorder(10,0,0,0));

//Lay out the text controls and the labels.
JPanel textControlsPane = new JPanel();
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
textControlsPane.setLayout(gridbag);

JLabel[] labels = {dataFieldLabel, commandFieldLabel};
JTextField[] textFields = {dataField, commandField};
addLabelTextRows(labels, textFields, gridbag,
textControlsPane);

c.gridwidth = GridBagConstraints.REMAINDER; //last
c.anchor = GridBagConstraints.WEST;
c.weightx = 1.0;
gridbag.setConstraints(actionLabel, c);
textControlsPane.add(actionLabel);
textControlsPane.

//Create a text pane.
textPane = createTextPane();
JScrollPane paneScrollPane = new JScrollPane(textPane);
paneScrollPane.setVerticalScrollBarPolicy(
    JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
//paneScrollPane.setHorizontalScrollBarPolicy(
    // JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
paneScrollPane.setPreferredSize(new Dimension(450, 355));
paneScrollPane.setMinimumSize(new Dimension(10, 10));

80
JPanel outputPane = new JPanel();
outputPane.setLayout(new GridLayout(1, 0));
outputPane.add(paneScrollPane);
outputPane.setBorder(BorderFactory.createCompoundBorder(
    BorderFactory.createTitledBorder("Watch Window"),
    BorderFactory.createEmptyBorder(5, 5, 5, 5)));

// buttons for setting up connection
JButton resetButton = new JButton("RESET!");
resetButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String s = term.reset();
        Document doc = textPane.getDocument();
        try {
            doc.remove(0, doc.getLength());
            doc.insertString(doc.getLength(), s + "\n",
                textPane.getStyle("bold"));
        } catch (BadLocationException ble) {
            System.err.println("Couldn't write watch window");
        }
    }
});

JButton autoConnectButton = new JButton("Slave Mode");
autoConnectButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String s = term.slaveMode();
        Document doc = textPane.getDocument();
        try {
            doc.insertString(doc.getLength(), s + "\n",
                textPane.getStyle("bold"));
        } catch (BadLocationException ble) {
            System.err.println("Couldn't write watch window");
        }
    }
});

JButton scanButton = new JButton("Inquire");
scanButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String s = term.inquire();
        Document doc = textPane.getDocument();
        try {
            doc.insertString(doc.getLength(), s + "\n",
                textPane.getStyle("bold"));
        } catch (BadLocationException ble) {
            System.err.println("Couldn't write watch window");
        }
    }
});
JButton connectButton = new JButton("Connect to:");
connectButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String s = term.connect(addressField.getText());
        Document doc = textPane.getDocument();
        try {
            doc.insertString(doc.getLength(), "connecting to "+
                addressField.getText() +"\n" + s + "\n",
            textPane.getStyle("bold"));
        } catch (BadLocationException ble) {
            System.err.println("Couldn't write watch window");
        }
    }
});
addressField = new JTextField("250aff5b0200");
JPanel connectPane = new JPanel();
connectPane.setLayout(new GridLayout(0, 1));
connectPane.add(connectButton);
connectPane.add(addressField);

JPanel buttonPane = new JPanel();
    buttonPane.setBorder(BorderFactory.createEmptyBorder(
        30, //top
        30, //left
        10, //bottom
        30) //right
    );
    buttonPane.setLayout(new GridLayout(1, 0));
    buttonPane.add(resetButton);
    buttonPane.add(autoConnectButton);
    buttonPane.add(scanButton);
    buttonPane.add(connectPane);

// buttons for polling feature
    JLabel scaleLabel = new JLabel("My Scale Factor");
    scaleField = new JTextField("3");
    JPanel scalePane = new JPanel();
    scalePane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10))
    ;
    scalePane.setLayout(new GridLayout(0,1));
    scalePane.add(scaleLabel);
    scalePane.add(scaleField);

    pollField = new JTextField("37");
    JButton pollButton = new JButton("Scale it!");
pollButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String request = "Req:" + pollField.getText();
        try {
            resultLabel.setText("");
            Integer.parseInt(pollField.getText());
            String s = term.sendData(request);
            Document doc = textPane.getDocument();
        } catch (NumberFormatException nfe) {
            System.err.println("Invalid number");
            return;
        }
    }
});
try {
    doc.insertString(doc.getLength(),
        s + "\n",
        textPane.getStyle("bold"));
}
catch (BadLocationException ble) {
    System.err.println("Couldn't write watch
    window");
}
} catch (NumberFormatException nfe) {
    pollField.setText("Invalid Number");
}
}

 JPanel pollPane = new JPanel();
   //pollPane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10))
   pollPane.
       setBorder(BorderFactory.
           createCompoundBorder(BorderFactory.
                createLineBorder(Color.black),
                BorderFactory.createEmptyBorder(3,3,3,3)));
    pollPane.setLayout(new GridLayout(0,1));
    pollPane.add(pollField);
    pollPane.add(pollButton);

    JPanel resultPane = new JPanel();
    JLabel resultLabelLabel = new JLabel("The response is:");
    resultLabel = new JLabel("");
    resultPane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
    resultPane.setLayout(new GridLayout(0, 1));
    resultPane.add(resultLabelLabel);
    resultPane.add(resultLabel);

    JPanel queryPane = new JPanel();
    queryPane.
        setBorder(BorderFactory.
            createCompoundBorder(BorderFactory.
                createTitledBorder("Querry Fields"),
                BorderFactory.
                    createEmptyBorder(5,5,5,5)));

    queryPane.setLayout(new GridLayout(1, 0));
    queryPane.add(scalePane);
    queryPane.add(pollPane);
    queryPane.add(resultPane);

    //Put everything in the applet.
    JPanel contentPane = new JPanel();
    BoxLayout box = new BoxLayout(contentPane, BoxLayout.Y_AXIS);
    contentPane.setLayout(box);
    contentPane.add(buttonPane);
    contentPane.add(textControlsPane);
}

83
```
contentPane.add(queryPane);
contentPane.add(outputPane);
setContentPane(contentPane);

private void addLabelTextRows(JLabel[] labels,
   JTextField[] textFields,
   GridBagLayout gridbag,
   Container container) {

   GridBagConstraints c = new GridBagConstraints();
   c.anchor = GridBagConstraints.EAST;
   int numLabels = labels.length;

   for (int i = 0; i < numLabels; i++) {
      c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last
      c.fill = GridBagConstraints.NONE; //reset to default
      c.weightx = 0.0; //reset to default
      gridbag.setConstraints(labels[i], c);
      container.add(labels[i]);

      c.gridwidth = GridBagConstraints.REMAINDER; //end row
      c.fill = GridBagConstraints.HORIZONTAL;
      c.weightx = 1.0;
      gridbag.setConstraints(textFields[i], c);
      container.add(textFields[i]);
   }
}

public void actionPerformed(ActionEvent e) {
   String prefix = "Sent ";
   if (e.getActionCommand().equals(dataFieldString)) {
      JTextField source = (JTextField)e.getSource();
      actionLabel.setText(prefix + "data " + source.getText() + 
      "\n");

      String s = term.sendData(source.getText());
      Document doc = textPane.getDocument();
      try {
         doc.insertString(doc.getLength(), s + "\n",
            textPane.getStyle("bold"));
      } catch (BadLocationException ble) {
         System.err.println("Couldn't insert initial text.");
      }
      source.setText("\n");
   }

   if (e.getActionCommand().equals(commandFieldString)) {
      JTextField source = (JTextField)e.getSource();
      actionLabel.setText(prefix + "command " + 
      source.getText() + "\n");
      String s = term.sendCommand(source.getText());
      Document doc = textPane.getDocument();
      try {
         doc.insertString(doc.getLength(), s + "\n",
            textPane.getStyle("bold"));
      }
   }
}
```
catch (BadLocationException ble) {
    System.err.println("Couldn't insert initial text.");
}

source.setText("\n");

private JTextPane createTextPane() {
    JTextPane textPane = new JTextPane();
    initStylesForTextPane(textPane);

    Document doc = textPane.getDocument();
    try {
        doc.insertString(doc.getLength(), serialPortStatus ,
                        textPane.getStyle("blue"));
    }
    catch (BadLocationException ble) {
        System.err.println("Couldn't insert initial text.");
    }

    return textPane;
}

protected void initStylesForTextPane(JTextPane textPane) {
    //Initialize some styles.
    Style def = StyleContext.getDefaultStyleContext().
            getStyle(StyleContext.DEFAULT_STYLE);

    Style regular = textPane.addStyle("regular", def);
    StyleConstants.setFontFamily(def, "SansSerif");

    Style s = textPane.addStyle("italic", regular);
    StyleConstants.setItalic(s, true);
    StyleConstants.setForeground(s, Color.red);

    s = textPane.addStyle("bold", regular);
    StyleConstants.setBold(s, true);
    StyleConstants.setForeground(s, Color.blue);

    s = textPane.addStyle("small", regular);
    StyleConstants.setFontSize(s, 10);

    s = textPane.addStyle("large", regular);
    StyleConstants.setFontSize(s, 16);

    s = textPane.addStyle("blue", regular);
    StyleConstants.setForeground(s, Color.blue);
}

public static void main(String[] args) {
    term = new Terminal();
    String comport;
    int baudRate;
    if (args.length == 2) {
        comport = args[0];
baudRate = Integer.parseInt(args[1]);
} else {
    comport = "COM1";
    baudRate = 115200;
}
try{
    term.start(comport, baudRate);
    serialPortStatus = "COM port opened successfully\n";
    serialPortStatus += term.reset();
} catch (SerialPortSetupException e) {
    System.err.println("Failed to open serial port.\n");
    System.err.println(e);
}
JFrame frame = new BlueGUI();
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
frame.pack();
frame.setVisible(true);

Document doc = textPane.getDocument();
String result = "";
while(true==true) {
    try {
        result = term.read();
        if(result.length() > 0) {
            doc.insertString(doc.getLength(), result,
            textPane.getStyle("italic"));
            int i = result.indexOf("Req:" );
            String s = "";
            if(i > 0) {
                s = result.substring(i + 4);
                try {
                    int value = Integer.parseInt(s);
                    int scale = Integer.parseInt(scaleField.getText());
                    s = s + "x" + scaleField.getText() + "=" + (value*scale);
                    s = term.sendData("Ans:" + s); 
                    doc.insertString(doc.getLength(), s,
                    textPane.getStyle( "bold"));
                } catch(Exception e) {}
            }
            i = result.indexOf("Ans:" );
            if(i > 0) {
                s = result.substring(i + 4);
                resultLabel.setText(s);
            }
        }
    } catch (BadLocationException ble) {
        System.err.println("Couldn't insert text.");
    } catch(Exception e) {}
}
try {
    Thread.sleep(100);
} catch (Exception e) {}
Appendix C: Source code for MSP430 based Bluetooth

This appendix contains a simple assembly-level program for the TI-MSP430f149 that controls a Bluetooth module. It was written for the IAR assembler available from Texas Instruments. The first file BluetoothDemoADCLED.s43 shows a fairly complete test setup for using the MSP430 as the controller of a Bluetooth slave device. The second file BluetoothMaster.s43 is a simple program to control a master device.

**BluetoothDemoADCLED.s43**

```assembly
#include "msp430x14x.h"

;*****************************************************************************
;       works in autonomous slave mode 
;       sets up slave mode -> turns on LED when ready for connect 
;       after connect turns of LED sends "Hello" and "World" turns on LED 
;       listens to bluetooth module 
;       when it receives a data packet with one byte in it: stores that byte in R10 
;       if the byte in R10 is 0x37 (ascii 7) sends ADC data back over 
;       bluetooth 
;       if R10=F(0x46) turns LED off 
;       if R10=N(0x4E) turns LED on 
;       deals with longer incoming data strings (but only ignores them) 
;*****************************************************************************

;UART0_TX_PTR DS 0x2
;UART0_TX_CNT DS 0x2 ;255 byte maximum in transmit string

ORG 0200h

UART0_TX_PTR DS 0x2
UART0_TX_CNT DS 0x2 ;255 byte maximum in transmit string
UART0_RX_INC DS 0x2 ; stores increment postion in bluetooth receive buffer
UART0_RX_BUF DS 0x256 ;256 bytes in receive buffer
PACKET_LENGTH DS 0x1 ; 1 byte for packet length
PACKET_BUF DS 0x64 ; 64 byte packet for the moment

org 0200h

RESET mov.w #0A00h,SP ; Initialize stackpointer
call #Init_Sys 
Mainloop mov.w #0x0, UART0_RX_INC ; reset increment in buffer
```

88
call #BT_RESET
Reset cmp #0x07, UART0_RX_INC ; wait for 7 characters
(minimum reset response)
jn Reset

mov.w #0x00, UART0_RX_INC ; reset increment in buffer
call #BT_AUTO_ACCEPT_CON_SL1
Slave1a cmp #0x07, UART0_RX_INC ; wait for 7 characters
(minimum connect response)
jn Slave1a

mov.w #0x00, UART0_RX_INC ; reset increment in buffer
call #BT_WRITE_SCAN_ENABLE_SL2
Slave1b cmp #0x07, UART0_RX_INC ; wait for 21 characters
(minimum connect response)
jn Slave1b

mov.w #0x00, UART0_RX_INC ; reset increment in buffer
call #BT_AUTO_ACCEPT_CON_SL1
Slave2a cmp #0x07, UART0_RX_INC ; wait for 7 characters
(minimum connect response)
jn Slave2a

mov.w #0x00, UART0_RX_INC ; reset increment in buffer
call #BT_WRITE_SCAN_ENABLE_SL2
Slave2b cmp #0x07, UART0_RX_INC ;
jn Slave2b

mov.b #0x01,&P1OUT ; turn on light
mov.w #0x00, UART0_RX_INC ; reset increment in buffer

ConnectionWait cmp #0x14, UART0_RX_INC ; wait for 21 characters
(minimum connect response)
; NOTE 0x15 but it is yet i get 14... where does buffer start... isn't right for this
jn ConnectionWait

mov.w #0x00, UART0_RX_INC ; reset increment in  

mov.b #0x00, &P1OUT ; turn off light

; move "Hello!" into packet buffer and transmit
mov.w #PACKET_BUF, R9 ; load pointer into R9
mov.b #0x48,0(R9)
mov.b #0x65,1(R9)
mov.b #0x6c,2(R9)
mov.b #0x6c,3(R9)
mov.b #0x6f,4(R9)
mov.b #0x21,5(R9)
mov.b #0x6, PACKET_LENGTH ; store length of Hello!
call #BT_PACKET_TRANSMIT;
Sent_data1 cmp #0x08, UART0_RX_INC ; wait for 8 characters
(jsend response)
    jn Sent_data1
    mov.w #0x0, UART0_RX_INC ; reset increment in
receive buffer

; move "World" into packet buffer and transmit
    mov.w # PACKET_BUF, R9 ; load pointer into R9
    mov.b #0x57,0(R9)
    mov.b #0x6f,1(R9)
    mov.b #0x72,2(R9)
    mov.b #0x6c,3(R9)
    mov.b #0x64,4(R9)
    mov.b #0x5, PACKET_LENGTH ; store length of Hello!
    call #BT_PACKET_TRANSMIT
Sent_data2 cmp #0x08, UART0_RX_INC ; wait for 8 characters
(msend response)
    jn Sent_data2
    mov.w #0x0, UART0_RX_INC ; reset increment in
receive buffer

    mov.b #0x01,&P1OUT ; turn on light
data
    clr.w R14 ; R14 is where I put ADC
    control register
    mov.w #0x00, UART0_RX_INC ; reset increment in
receive buffer
Mainloop1 cmp #0x4E, R10 ; 0x4E = N for on turn on led
    jnz NoLight
    mov.b #1h,&P1OUT
NoLight cmp #0x46, R10 ; 0x46 = F for off turn on led
    jnz Light
    mov.b #0h,&P1OUT
Light cmp #0x37, R10
    jnz Mainloop1 ; if in mode 0x37
(acquisition) get data
; Take ADC reading on chan 0 place in packet buffer,
transmit 8 samples
; over Bluetooth
    mov.w #0x00, UART0_RX_INC ; reset increment in
receive buffer

;;;;; This is from TI's demo code
    bis #ADC12SC,&ADC12CTL0 ; start A/D conversion
testEOC bit #BITO,&ADC12IFG ; end of convert?
(ADC12IFG.0=1?)
    jz testEOC ; no, test again
    mov &ADC12MEM0,R4 ; yes, result moved to R4,
flag
    ; is automatically cleared
;;;; end of TI demo code
    mov R14, R13
    add #PACKET_BUF, R13
mov    R4, 0(R13)          ; increment counter
add    #0x2, R14          ; each sample is a word
cmp    #0x10, R14         ; done acquiring

measurements?
clr.w  R14                ; send packet over Bluetooth
mov.b  #0x10, PACKET_LENGTH ; 16 bytes in 8 samples
call   #BT_PACKET_TRANSMIT

Sent_data cmp    #0x08, UART0_RX_INC ; wait for 8 characters
(minimum send response)

jn      Sent_data
mov.w  #0x0, UART0_RX_INC ; reset increment in
receive buffer
NotDone jmp   Mainloop1 ; Start measuring again

; Init_Sys; Initialize MSP430 system
;----------------------------------------------------------------------

StopWDT  mov.w  #WDTPW+WDTHOLD,&WDTCTL        ; Stop WDT
SetupBC  bis.b  #XTS,&BCSCTL1                 ; LFXT1 = HF XTAL
         clr.w  UART0_RX_INC                   ; set receive counter to zero
SetupUART0 bis.b #UTXEO+URXEO,&ME1          ; Enable USART0 TXD/RXD
         mov.b  #CHAR,&UCTL0                   ; 8-bit characters, no parity, 1 stop
         mov.b  #022h,&UBR00                    ; Bluetooth Mode
         mov.b  #000h,&UBR10                    ; 4Mhz/115200 = 34.72222
         clr.b  &UMCTL0                        ; 4Mhz no modulation 115200

IS THIS SAFE?  
bis.b  #URXIE0,&IE1        ; Enable USART0 RX interrupt

SetupP3  bis.b  #030h,&P3SEL                  ; Select USART1/0 TXD/RXD
         bis.b  #010h,&P3DIR                   ; P3.6,4 = output direction
SetupLED bis.b  #001h,&P1DIR                  ; LED pin for testing
         mov.b  #0h,&P1OUT                    ; turn of LED initially

;------------------------Start of ADC section------------------------
SetupADCl2 mov  #SHP+ADCl2SSEL_2,&ADC12CTL1        ; use sample timer, ADC core
              ; runs at MLCK, single
channel mov.b  #00h,&ADC12MCTL0                  ; end of sequence(EOS)=0 - is not
          chan. mode,
=AVss, input bis.b  #01h,&P6SEL                  ; channel A0
          ; Vref+ =AVcc, Vref-
          ; set Port6.0 pin to ADC

function
; NOTE: loading of ADC12MCTRL0 not actually required in this example since it
; powers up as all zeros, just included here as a reminder

    mov   #REFON+ADC12ON+ENC,&ADC12CTRL0 ; sample time x1, single
        ; sample

; NOTE: ADC12CTRL0 set ENC last, since set ENC bit prevents some bits in
; ADC12CTRL1, ADC12CTRL0 & ADC12MCTRLx from being changed

    bis   #ENC,&ADC12CTRL0 ; set ENC bit

;--------------------- END of ADC section---------------------

    eint ; General enable interrupts
    ret ; Return from subroutine

;--------------------- BlueTooth routines---------------------

BT_TEST ; sends "Hello!" to RS232
    mov.w   #BT_TEST_STR, UART0_TX_PTR ; move reset string pointer into place
    mov.w   #0x06, UART0_TX_CNT ; 6 bytes in Hello!
    call    #ARRAY_TRANSMIT
    ret

BT_RESET ; sends a reset to CASIRA
    mov.w   #BT_RESET_STR, UART0_TX_PTR ; move reset string pointer into place
    mov.w   #0x04, UART0_TX_CNT ; 4 bytes in reset
    call    #ARRAY_TRANSMIT
    ret

BT_CONNECT ; sends connect request
    mov.w   #BT_CONNECT_STR, UART0_TX_PTR ; move reset string pointer into place
    mov.w   #0x11, UART0_TX_CNT ; 17 bytes in reset
    call    #ARRAY_TRANSMIT
    ret

; the two slave commands
BT_AUTO_ACCEPT_CON_SL1
    mov.w   #BT_AUTO_ACCEPT_CON_SL1_STR, UART0_TX_PTR ; move reset string pointer into place
    mov.w   #0x07, UART0_TX_CNT
    call    #ARRAY_TRANSMIT
    ret

; inquiry and page scan on:
BT_WRITE_SCAN_ENABLE_SL2
    mov.w   #BT_WRITE_SCAN_ENABLE_SL2_STR, UART0_TX_PTR ; move reset string pointer into place
    mov.w   #0x05, UART0_TX_CNT
    call    #ARRAY_TRANSMIT
    ret
BT_PACKET_TRANSIT ;sends PACKET_LENGTH bytes from PACKET_BUF after wrapping in appropriate
; Bluetooth header
; Trashes R8
    mov.w  #BT_DATA_PACKET_HEADER_STR, UART0_TX_PTR ;move
pointer to first part of transmit
    mov.w  #0x03, UART0_TX_CNT
    call   #ARRAY_TRANSIT

    Pack_TX1 bit.b #UTXIFGO,&IFG1
           jz    Pack_TX1 ; USART0 TX buffer ready?
     jz    Pack_TX2 ; Jump is TX buffer not ready
     mov.b PACKET_LENGTH, &TXBUF0 ; transmit LSB of length
     bit.b #UTXIFGO,&IFG1 ; USART0 TX buffer ready?
     mov.w UART0_TX_PTR, R5 ; R5 is pointer
     mov.w UART0_TX_CNT, R6 ; R6 will be upper bound
     add.w R5, R6 ; R6 = ptr + cnt
     TX_ARR bit.b #UTXIFGO,&IFG1 ; USART0 TX buffer ready?
     jz    TX_ARR ; Jump is TX buffer not ready
     mov.b @R5+, &TXBUF0 ;transmit next byte of array
     cmp.w R6, R5
     jl    TX_ARR ; if R5!=number of bytes to send: loop
    ret

ARRAY_TRANSIT ;transmits a string of bytes over the serial port
; Trashes: R5, R6
; This routine depend on having the pointer to the command string in
; UART0_TX_PTR, and the number of bytes to be sent in UART0_TX_CNT
    mov.w UART0_TX_PTR, R5 ; R5 is pointer
    mov.w UART0_TX_CNT, R6 ; R6 will be upper bound
    add.w R5, R6 ; R6 = ptr + cnt
    TX_ARR bit.b #UTXIFGO,&IFG1 ; USART0 TX buffer ready?
    jz    TX_ARR ; Jump is TX buffer not ready
    mov.b @R5+, &TXBUF0 ;transmit next byte of array
    cmp.w R6, R5
    jl    TX_ARR ; if R5!=number of bytes to send: loop
    ret

CHECK_INPUT ; If an incoming string of bytes is a data packet (starts with 0x02)
; store a one byte payload in R10 deal with longer payloads
; TRASHES: R11, R12 (only sometimes), R10 <- don't save r10 it's the control register
; NOTE: assumes data packets < 256 bytes
; if we start using commands longer than 1 byte may want to move the moreData
; function up a ways
;
; In general if the microcontroller gets out of sync (gets an unexpected command)
; it's dead (buffer will get messed up)
; In the long run it would be good to actually examine incoming data packets
; by type and extract length info.

93
mov #UART0_RX_BUF, R11 ; R11 is pointer into receive buffer

cmp.b #0x05, UART0_RX_INC ; are there at least 6 bytes (use 0x05 because receive pointer hasn't been incremented yet) in the data?
    jl EndChkIn ; less than 5 bytes: punt
    ; the next two checks are probably extraneous we only make connections to handle 2600
    ; of course if that handle changes could be bad... store handle somewhere on connect
    cmp.b #0x26, 1(R11) ; are the next two bytes the rest of the header
        jnz EndChkIn
    cmp.b #0x20, 2(R11) ; are the next two bytes the rest of the header
        jnz EndChkIn
    cmp.b #0x00, 4(R11) ; MSB length
        jnz EndChkIn ; ERROR!!!!
    cmp.b #0x01, 3(R11)
        jnz MoreData

    ; if it's a one byte command: store it in R10
    mov.b 5(R11), R10 ; R10 holds incoming command

    mov.w #-1, UART0_RX_INC ; clear the receive buffer (will be incremented on exit from receive interrupt)
EndChkIn ret

; if there is more than one byte in the incoming data packet see if the entire packet has been received and then deal with it (in this case just clear the RX buffer)
MoreData mov.b UART0_RX_INC, R12 ; length of received data
sub.b #0x04, R12 ; subtract out header (5 bytes so use 4 because UART0_RX_INC starts at 0) for data packet
    cmp.b #0x00, 3(R11) ; compare data packet length to received data
        jnz EndChkIn2 ; Deal with complete data packet
mov.w #-1, UART0_RX_INC ; finished receiving packet clear the receive buffer (will be incremented on exit from receive interrupt)
EndChkIn2 ret

-----------------------------------------------

-----

USART_ISR; Receive character, place in buffer

-----

; TRASHES: R7
; This routine depends on UART0_RX_BUF being a pointer to the start of a receive buffer
; UART0_RX_INC holds byte increment in buffer
    mov.w #UART0_RX_BUF, R7 ; move pointer into R5
    add.w UART0_RX_INC, R7 ; add increment into R5-
>position in buffer
    mov.b &RXBUF0, 0(R7) ; copy rs232 into memory
loc
data?
Correctly parsing packets
jnz NoChkCom
No need to check packet
add.w #0x01, UART0_RX_INC
buffer reti
from interrupt

; Definitions for bluetooth commands
---
BT_TEST_STR DB 0x48,0x65,0x6c,0x6c,0x6f,0x21 ; Hello!
BT_RESET_STR DB 0x01,0x03,0x0c,0x00 ; 4 bytes
BT_READ_BD_ADDR_STR DB 0x01,0x09,0x10,0x00 ; 4 bytes
BT_CONNECT_STR DB 0x01,0x05,0x04,0x0d,0x25,0x0a,0xff,0x5b,0x02,0x00,0xc0,0x18,0x01,0x00,0
x00,0x00,0x00; 17 bytes hard code 0000 as clock offset has always worked okay
BT_DATA PACKET HEADER STR DB 0x02,0x26,0x20 ; 3 bytes
BT_AUTO_ACCEPT_CON SL1 STR DB 0x01,0x05,0x0c,0x03,0x02,0x00 ; 7 bytes
BT_WRITE_SCAN_ENABLE SL2 STR DB 0x01,0x1a,0x0c,0x01,0x03; 5 bytes
---
Interrupt Vectors Used MSP430x13x/14x
---
ORG OFFFEh
DW RESET ; POR, ext. Reset, Watchdog
ORG OFFF2h
DW USARTISR ; USART0 receive; JOHANN:
jump to USART_ISR on interrupt
end

BluetoothDemoMaster.s43

include "msp430x14x.h"
;*******************************************************************************
*******
; works in autonomous mode
;*******************************************************************************
*******
; UART0_TX_PTR DS 0x2
; UART0_TX_CNT DS 0x2 ; 255 byte maximum in transmit string
ORG 0200h
UART.TXT_PTR DS 0x2
UART.TXT_CNT DS 0x2 ; 255 byte maximum in transmit string
UART.RX_INC DS 0x2 ; stores increment position in Bluetooth receive buffer
UART.RX_BUF DS 0x256 ; 256 bytes in receive buffer
PACKET_LENGTH DS 0x1 ; 1 byte for packet length
PACKET_BUF DS 0x64 ; 64 byte packet for the moment

-----------------------------
ORG 0F000h ; Program Start
-----------------------------
RESET    mov.w #0A00h, SP    ; Initialize stack pointer
call #Init_Sys
Mainloop mov.w #0x0, UART0_RX_INC ; reset increment in buffer
call #BT_RESET
Reset    cmp #0x07, UART0_RX_INC ; wait for 7 characters
  (minimum reset response)
jn Reset
  mov.w #0x0, UART0_RX_INC ; reset increment in buffer
Connect  call #BT_CONNECT
   cmp #0x07, UART0_RX_INC ; wait for 7 characters
  (minimum connect response)
jn Connect
    mov.w #0x0, UART0_RX_INC ; reset increment in buffer
Connect2 call #BT_CONNECT
   cmp #0x15, UART0_RX_INC ; wait for 21 characters
  (minimum connect response)
jn Connect2
    mov.w #0x0, UART0_RX_INC ; reset increment in buffer

Sent_datal mov.b #0x01, &P1OUT ; turn on light

; move "Hello!" into packet buffer and transmit
mov.w #PACKET_BUF, R9  ; load pointer into R9
mov.b #0x48, 0(R9)
mov.b #0x65, 1(R9)
mov.b #0x6c, 2(R9)
mov.b #0x6c, 3(R9)
mov.b #0x6f, 4(R9)
mov.b #0x21, 5(R9)
mov.b #0x6, PACKET_LENGTH ; store length of Hello!
call #BT_PACKET_TRANSMIT;

Sent_datal cmp #0x08, UART0_RX_INC ; wait for 8 characters
  (send response)
jn Sent_datal

96
mov.w #0x0, UART0_RX_INC ; reset increment in receive buffer

; move "World" into packet buffer and transmit
mov.w #PACKET_BUF, R9 ; load pointer into R9
mov.b #0x57,0(R9)
mov.b #0x6f,1(R9)
mov.b #0x72,2(R9)
mov.b #0x6c,3(R9)
mov.b #0x64,4(R9)
mov.b #0x5, PACKET_LENGTH ; store length of Hello!
call #BT_PACKET_TRANSMIT
Sent_data2 cmp #0x08, UART0_RX_INC ; wait for 8 characters
(msend response)
jn Sent_data2
mov.w #0x0, UART0_RX_INC ; reset increment in receive buffer

clr.w R14
Mainloop1 ; Take ADC reading on chan 0 place in packet buffer
transmit 8 sample
; over Bluetooth
mov.w #0x0, UART0_RX_INC ; reset increment in receive buffer

;;;;; This is from TI's demo code
bis #ADC12SC,&ADC12CTL0 ; start A/D conversion
testEOC bit #BIT0,&ADC12IFG ; end of convert?
(ADC12IFG.0=1?) 
jz testEOC ; no, test again
mov &ADC12MEMO,R4 ; yes, result moved to R4, flag

;;;;; end of TI demo code
mov R14, R13
add #PACKET_BUF, R13
mov R4, 0(R13)
add #0x2, R14 ; increment counter
cmp #0x10, R14 ; each sample is a word
jn NotDone ; done acquiring measurements?
clr.w R14 ; send packet over Bluetooth
mov.b #0x10, PACKET_LENGTH ; 16 bytes in 8 samples
call #BT_PACKET_TRANSMIT
Sent_data cmp #0x08, UART0_RX_INC ; wait for 8 characters
(minimum send response)
jn Sent_data
mov.w #0x0, UART0_RX_INC ; reset increment in receive buffer
NotDone jmp Mainloop1 ; Start measuring again

;------------------------------------------------------------------------
Init_Sys; Initialize MSP430 system
;------------------------------------------------------------------------

97
StopWDT    mov.w  #WDTPW+WDTHOLD,&WDTCTL ; Stop WDT
SetupBC    bis.b  #XTS,&BCSCTL1  ; LFXT1 = HF XTAL
clr.w     UART0_RX_INC  ; set receive counter to zero
SetupUART0 bis.b  #UTXE0+URXE0,&ME1 ; Enable USART0 TXD/RXD
mov.b    #CHAR,&UCTL0  ; 8-bit characters, no parity, 1 stop->BlueTooth Mode
        mov.b  #SSEL0,&UTCTL0 ; UCLK = ACLK
        mov.b  #022h,&UBR00  ; 4Mhz/115200 = 34.72222
        mov.b  #000h,&UBR10  ; 4Mhz 115200
clr.b    &UMCTL0  ; 4MHz no modulation 115200

IS THIS SAFE?
        bis.b  #URXIE0,&IE1  ; Enable USART0 RX
interrupt
SetupP3     bis.b  #030h,&P3SEL  ; Select USART1/0 TXD/RXD
        bis.b  #010h,&P3DIR  ; P3.6,4 = output direction
SetupLED     bis.b  #001h,&P1DIR  ; LED pin for testing
        mov.b  #0h,&PIOUT  ; turn of LED initially

;------------------- Start of ADC section-------------------
SetupADC12 mov    #SHP+ADC12SSEL_2,&ADC12CTL1 ; use sample timer, ADC core
channel
        mov.b  #00h,&ADC12MCTL0  ; end of sequence (EOS)=0 - is not
        ; required for single chan. mode,
        ; Vref+ =AVcc, Vref- =AVss, input

        bis.b  #01h,&P6SEL  ; set Port6.0 pin to ADC function

; NOTE: loading of ADC12MCTL0 not actually required in this example since it
; powers up as all zeros, just included here as a reminder
        mov    #REFON+ADC12ON+ENC,&ADC12CTL0 ; sample time x1, single

; NOTE: ADC12CTL0 set ENC last, since set ENC bit prevents some bits in
; ADC12CTL1, ADC12CTL0 & ADC12MCTLx from being changed
        bis    #ENC,&ADC12CTL0  ; set ENC bit
;--------------------- END of ADC section---------------------
eint  ; General enable interrupts
ret   ; Return from subroutine

;----------------------- BlueTooth routines------------------------
;BTTEST ;sends "Hello!" to RS232
mov.w  #BTTEST_STR, UART0_TX_PTR ;move reset string
pointer into place
mov.w  #0x06, UART0_TX_CNT ;6 bytes in Hello!
call  ARRAY_TRANSMIT
ret

BT_RESET ;sends a reset to CASIRA
mov.w  #BT_RESET_STR, UART0_TX_PTR ;move reset string
pointer into place
mov.w  #0x04, UART0_TX_CNT ;4 bytes in reset
call  ARRAY_TRANSMIT
ret

BTCONNECT ;sends connect request
mov.w  #BTCONNECT_STR, UART0_TX_PTR ;move reset string
mov.w  #0x11, UART0_TX_CNT ;17 bytes in reset
call  ARRAY_TRANSMIT
ret

BT_PACKETTRANSMIT ;sends PACKET_LENGTH bytes from PACKET_BUF after wrapping in appropriate
;Bluetooth header
;Trashes R8
mov.w  #BT_DATA_PACKET_HEADER_Str, UART0_TX_PTR ;move
pointer to first part of transmit
mov.w  #0x03, UART0_TX_CNT
call  ARRAY_TRANSMIT
Pack_TX1 bit.b  #UTXIFG0,&IFG1 ;USART0 TX buffer ready?
jz    Pack_TX1 ;Jump is TX buffer not ready
mov.b PACKET_LENGTH, &TXBUF0 ;transmit LSB of length
Pack_TX2 bit.b  #UTXIFG0,&IFG1 ;USART0 TX buffer ready?
jz    Pack_TX2 ;Jump is TX buffer not ready
mov.b #0x00, &TXBUF0 ;transmit MSB of length
mov.w  #PACKET_BUF, UART0_TX_PTR ;contents
mov.b  #PACKET_LENGTH, UART0_TX_CNT ;length of packet
call  ARRAY_TRANSMIT
ret

ARRAY_TRANSMIT ;transmits a string of bytes over the serial port
;Trashes: R5, R6
;This routine depend on having the pointer to the command string in
;UART0_TX_PTR, and the number of bytes to be sent in UART0_TX_CNT
mov.w  UART0_TX_PTR, R5 ;R5 is pointer
mov.w  UART0_TX_CNT, R6 ;R6 will be upper bound
add.w  R5, R6 ;R6 = ptr + cnt
TX_ARR bit.b  #UTXIFG0,&IFG1 ;USART0 TX buffer ready?
jz    TX_ARR ;Jump is TX buffer not ready
mov.b @R5+, &TXBUF0 ;transmit next byte of array
cmp.w  R6, R5
j1       TX_ARR           ; if R5! = number of bytes to
send: loop
ret

-----
USART_ISR; Receive character, place in buffer

-----
; TRASHES: R7
; This routine depends on UART0_RX_BUF being a pointer to the start of
a receive buffer
; UART0_RX_INC holds byte increment in buffer
mov.w   #UART0_RX_BUF, R7   ; move pointer into R5
add.w   UART0_RX_INC, R7   ; add increment into R5-
>position in buffer
mov.b   &RXBUF0, 0(R7)      ; copy rs232 into memory
loc     add.w   #0x01, UART0_RX_INC
         ; JOHANN: reti = return
buffer
reti
from interrupt

-----
; Definitions for bluetooth commands

-----
BT_COMS
BT_TEST_STR   DB  0x48,0x65,0x6c,0x66,0x21  ;Hello!
BT_RESET_STR  DB  0x01,0x03,0x0c,0x00    ; 4bytes
BT_READ_BD_ADDR_STR DB  0x01,0x09,0x10,0x00  ; 4 bytes
BT_CONNECT_STR DB  0x00,0x01,0x05,0x04,0x0d,0x25,0x0a,0xff,0x5b,0x02,0x00,0x0c,0x18,0x01,0x00,0
x00,0x00,0x00; 17bytes hard code 0000 as clock offset has always worked
okay
BT_DATA_PACKET_HEADER_STR DB  0x02,0x26,0x20    ; 3bytes
;the two slave commands
BT_AUTO_ACCEPT_CON SL1   DB  0x01,0x05,0x0c,0x03,0x02,0x00,0x02
; inquiry and page scan on:
BT_WRITE_SCAN_ENABLE_SL2 DB  0x01,0x0a,0x0c,0x10,0x03

-----
; Interrupt Vectors Used MSP430x13x/14x

-----
ORG     0FFFEh             ; POR, ext. Reset, Watchdog
DW      RESET
ORG     0FFFF2h
DW      USART_ISR
jump to USART_ISR on interrupt
end