An Infrared Activity Monitoring System

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)

by
Derek Brader

Project Advisor: Dr. Bruce R. Land
Degree Date: August 2003

Abstract

Master of Electrical and Computer Engineering Program

Cornell University

Design Project Report

Project Title: An Infrared Activity Monitoring System

Author: Derek Brader

Abstract: An infrared monitoring system was designed for use in the lab of Dr. Howard Howland. Through a series of experiments on young chickens, Dr. Howland is studying the development and control of eyes and vision. To conduct these experiments, a device is required which is capable of measuring the relative level of activity in a number of chicken cages. Infrared light beams are shown through the cages, and the level of activity can be approximated by the number of times the beams are broken. At the heart of this configurable and adaptable system is an Atmel 90S8515 microcontroller. Data is stored in the microcontroller's memory, and can be downloaded to a personal computer over an RS-232 serial data link. The goals of this project include ease of use, cost effectiveness, and the ability to configure and generalize the system.

Report Approved by

Project Advisor: _____ Date: _____

Executive Summary

Some experiments must be monitored so continuously than is possible for a human observer. While commercial automated monitoring systems can be purchased, a perfectly suited one may be very difficult to find, particularly if cost is a concern. Dr. Howard Howland, in Neurobiology and Behavior, is in need of such a system. Dr. Howland is studying the impact of an animal's environment on the development of its eyes and vision. With young chickens as experimental subjects, he wishes to monitor the level of activity in a number of cages at regular time intervals. This report details the design of a system which meets this need.

Activity can be measured with reasonable accuracy using a series of infrared beams. These beams can be arranged so that a few beams cross each cage. The activity monitor includes twelve such sensors, pairing a bright directional infrared light emitting diode with a photo-transistor detection unit. More activity in the cage will cause the beams to be broken more frequently. By counting the number of beam breaks in a time interval, the activity in that time interval can be accurately estimated.

This system was designed around an Atmel 90S8515 microcontroller with 32kB of external RAM. A user interface was designed with four pushbuttons and a serial-controlled LCD with graphical capabilities, allowing the user to easily configure and calibrate the system. The monitor efficiently stores a large amount of activity data in its memory until that data is downloaded to a personal computer via an RS-232 serial connection.

All planned features were implemented satisfactorily with no major problems. The monitor's interfaces were kept simple and standard, making it easy to adapt the monitor to alternate applications. Preliminary tests yielded very promising results, and the final product is just beginning to enter service as of this writing.

Table of Contents

I) Introduction

I.1) Motivation

It is frequently necessary for research scientists to take measurements over long periods of time. This could be accomplished by a human observer, but that would be a difficult and tedious task. People get tired, they make mistakes, and (perhaps most importantly) they are expensive. An automated monitoring system can be an excellent solution to this sort of problem, taking data on perfectly regular intervals for later analysis by the researcher.

Many data acquisition products are available on the market, but they tend to be quite expensive. Finding a suitable system may also prove to be difficult for certain applications. If a researcher has the proper resources, or access to them, designing and manufacturing a custom monitoring system could be an advantageous alternative. This is the case for Dr. Howard Howland's research.

I.2) Dr. Howland's Research

Dr. Howland is studying the development and control of the eye and vision. His research studies the effects of the environment on this development. Chickens are good subjects for this study since they are cheap, readily available, and develop relatively quickly. Dr. Howland wishes to measure the relative level of activity for his chicken cages as he varies environmental conditions.

I.3) A General Design

While Dr. Howland's research provides motivation for a very interesting design exercise, proper design could render the final product useful in other circumstances as well. By keeping this in mind during design, the final product could later be tailored for use in other applications. Doing this involves keeping the features generic, and keeping the interfaces simple and standard.

II) Design Problem and System Of Requirements

II.1) Design Problem

The goal of this project is to design a monitoring system which is to accurately measure the level of activity for four sets of chickens. Activity is to be measured and recorded on regular intervals over a long period of time, and should be accessible at a later date for analysis with any software of choice. Measurements are to be taken from four cages occupying two rooms.

It is important to decide on a metric for activity, defined in such a way that it can be easily measured and recorded by an electrical system. The need for a simple, robust system with minimal impact on the experiment led to a design using infrared beams. By arranging a series of beams that cross through the cages, we can define activity as the number of beam interruptions per unit time.

The activity monitor, then, is a device which counts beam breaks over a consecutive series of time intervals. Smaller time intervals will result in higher time-resolution in activity measurement. Dr. Howland will be interested in measuring activity at time intervals of about 10 minutes, so that activity can be accurately expressed for different times of day.

II.2) Requirements

In defining the requirements for any system, the designer must look primarily to the customer. He who will actually *use* the system will know best what attributes are important. Some discussion of the problem with Dr. Howland and Dr. Land led me to the following preliminary requirements:

- The system must perform its above described function (it must use infrared beams to measure activity on regular intervals, record its measurements, and make the resulting data available for spreadsheet analysis).
- A non-technical person must be able to effectively use the system.
- The system must not interfere with the experiment it is intended to measure.
- The system must comply with regulations for animal laboratories

This vague, preliminary set of requirements will be more explicitly specified in the sections to come.

III) Range of Solutions

The vague set of preliminary requirements outlined in the previous section do not specify a solution. Beyond those requirements, there are a number of design decisions which must be made in order to define a certain implementation. Several of these design decisions are best left to the discretion of the "customer." In appendix A, you will find the document I presented to Dr. Howland for him to consider the trade-offs involved in some of these design decisions.

III.1) User Interface

A well designed user interface is crucial if a system is to be used by anyone other than the designer. A good user interface could be designed any number of ways; I considered two options. One method would be to interface with the system using a computer. The computer could communicate serially with the monitoring system to set up various parameters and perform calibration, using either a generic terminal program or a custom application. The monitor itself would have a few lights to report its basic status. The second option would be to have the whole user interface available on the monitor itself. This would involve a liquid crystal display (LCD), a few push-buttons, and possibly a keypad.

III.2) Data Interface

Since the the goal of this device is to transfer data to a computer for later analysis, we must specify the way the data is to be transferred. Because of its wide availability and existing hardware, I looked no further than the RS-232 protocol for serial communication. Still remaining was the decision of how and when this communication channel was to be used. I considered two options.

The first option I termed "constant reporting" and requires a dedicated computer with a persistent serial connection to the monitor. This option does not require that the monitor have any memory at all, except perhaps to store configuration and working variables. It also relies more heavily on the stability of the computer than on that of the monitor: data could be lost due to a computer malfunction, but probably not

due to a monitor malfunction.

The second option I termed “interval reporting” and requires data to be downloaded to a computer periodically. This option requires the monitor to have enough memory that downloading data is not required too frequently. Too small a memory in this configuration would negate the point of using an automated system, since human intervention might so frequently be required to download the data. A monitor malfunction could result in data loss with this option, but it does not depend heavily on the stability of the computer. Probably the greatest strength of this option is that it does not require a dedicated computer, since computers are quite costly.

III.3) Controller/Memory Options

Which controller to use, and how much memory? More expensive controllers have more internal memory, more input/output ports, and more computing power. Along with their expense in dollars, the more complex controllers have a steeper learning curve, which would likely increase development time. At the same time, designing a complex system around a very simple controller could prove cumbersome. Choosing an appropriate controller is a delicate balance. The amount of memory required is an important issue, and is largely determined by the choice of reporting method described in section III.2.

III.4) Sensors

From this projects conception, it was assumed that infrared sensors were to be used. The use of such sensors was not questioned because they are fast, unobtrusive, widely used, and readily available at low cost.

A good method of mounting the sensors is necessary for a number of reasons. The system should be easy to use, so the mounting mechanism ought not be too cumbersome to set up. Room lighting may interfere with the sensors, so in order to get the needed range and sensitivity they must be mounted carefully. These sensors are to be used in on and around chicken cages, and chickens are not particularly clean creatures. The sensors must be protected from their environment. The sensors also must be small and unobtrusive such that they do not interfere with the experiments or the results.

Each sensor is actually a transmitter/receiver pair. I again considered two options for the way these two might communicate. One is for the receiver to be a simply detect the level of infrared light (IR), switching at a certain threshold. A more complicated, and more robust, solution would be to transmit some sort of carrier signal and detect that signal at the receiver. While this would be more difficult to implement, it would be far more immune to ambient light in the room.

We must also consider the interface between the sensors and the monitor itself. Keeping this interface standard would be vastly helpful if any improvements or changes are later to be made to the sensors. Ideally, almost any type of sensor could be used.

IV) Design and Implementation

Before much work at all could begin on the monitor, most of the design decisions outlined in section III above needed to be finalized.

IV.1) Reporting Method

It was decided to use the interval reporting method described above, without having a dedicated computer. This decision was heavily influenced by cost, and I believe it is a good one.

IV.2) User Interface

Since a dedicated computer would not available, the interface would need to be on the monitor itself. I decided to implement the user interface with four pushbuttons and an liquid crystal display (LCD). Chosen primarily because of its availability was a 4x20 character alphanumeric LCD with graphical capabilities. It takes display commands over a one-pin serial connection using a TTL-level RS-232 signal. The electronic requirements for this interface package are very attractive. It involves four input signals, one for each of the pushbutton switches, and one output signal to drive the LCD.

IV.3) Data Interface

The data interface is over an RS-232 connection at 9600 BAUD. A data transfer, initiated using the pushbuttons and LCD, will result in a dump of the monitor's memory as a comma separated list (CSV). This format is convenient because it is easy to generate, and can be easily imported into many popular analysis applications. Each line of the output corresponds to one data point, and contains the count for each sensor and a time/date stamp. For more information, see Appendix A, “Instructions For Use.”

IV.4) Controller / Memory

A balance of the requirements led to a decision to use the Atmel 90S8515

micro-controller. This 4 MHz controller features, among other things, four 8-bit input/output ports, one bidirectional serial UART, 512 bytes of random access memory (RAM), and 8kB of programmable Flash memory. If anything, the features of this controller are a bit lean; but that makes for a more interesting exercise in design.

As for the memory required, 32kB of external RAM is readily available for the selected controller; and this controller includes hardware support for the required memory interface. This is sufficient if the implementation is careful. If we assume that we will not generally see a count higher than 255 in any given interval, then we can store each count as an 8-bit unsigned integer. With twelve sensors, we will use 12 bytes for each data point, not including the required date stamp. If we "reconstruct" the date stamps at the time of upload, based on the data point index and the current time and date, then we relax the burden of storing them. 32kB / 12B = 2,730 data points. If we take a point every ten minutes, then we have enough memory to last nearly 19 days without losing data. We have enough for nearly two days if points are taken every minute, so this is clearly sufficient for the monitor's purpose.

The chosen Atmel microcontroller includes support for a serial Universal Asynchronous Transmit/Receive (UART) in hardware. When using a UART for communication, the complexity of the embedded application can be drastically reduced by relying on this hardware. Unfortunately, the chosen Atmel controller supports only one UART in hardware. Since the LCD as well as the data interface both rely on UART communication, I decided to implement the serial data transmitter for the LCD in software. Other standard LCD interfaces require much more than a single control signal, so the disadvantage of the added complexity to the embedded application was offset by the reduced requirement of output port pins.

IV.5) Sensors

Because the sensors are to be used in a very controlled environment, it is safe to choose the simpler implementation proposed above, to simply detect the intensity of IR currently incident on the receiver.

The sensors are mounted on a custom printed circuit board so that the

accompanying detection circuitry can be very close to the receiver, and so that the unit will be as compact as possible. The detection element is mounted in a blackened tube to mitigate the effects of ambient light. This structure will be housed in a small polystyrene box along with a connector for a cable to the monitor.

The detection element itself is a photo-transistor. An operational amplifier with open-loop gain and threshold set by a trim potentiometer drives the output to the monitor. A schematic for this circuit is included in Appendix E. The transmitter is a highly directional IR light emitting diode (LED). Experiments with this setup ensured that sufficient range can be achieved.

IV.6) Packaging, Cabling, and Power

Since only one unit is to be built, the circuitry for the monitor (apart from the sensors) is wire-wrapped. This was a time-consuming option, but offered considerable flexibility and cost savings over a custom printed circuit board. The monitor's chassis is an aluminum box, purchased from the physics stock-room. This was chosen for its cost, availability, and the ease with which it could be drilled and cut to accommodate the required components.

The issue of cabling between the monitor and the sensors really amounts to a price/quality trade-off. Since I, the designer, control neither the budget nor the use of the final product, I left this decision to the customer. A discussion of the cabling options considered can be found in my design proposal, included in Appendix A. It was decided that strength and durability outweighed the importance of cost, so we chose shielded cables with XLR connectors (which are commonly used for low-impedance stage and studio microphones). This option offers the required three conductors, two for power and one for the return signal. The shielding in these cables is probably unnecessary for the slow digital signals they will carry, but both the cables and connectors are physically robust and readily available at reasonable prices. One female XLR jack is mounted on each sensor, and 12 male jacks are mounted on the monitor's chassis.

Power for the monitor ultimately comes from the public power grid: it plugs into the wall. A small 9V DC power supply was purchased at what was probably a lower

cost than would have been required to make it from discrete components. All circuitry in the monitor runs on 5V DC. A heat-sinked regulator and a large number of distributed bypass capacitors on the wire-wrap board control voltage to the microcontroller and all other circuitry housed within the monitor's chassis. 9 volts travels across the shielded cables to the sensors, where it is locally regulated to 5V. Though this offers a very small efficiency advantage because of ohmic losses in the cables, the primary reason for doing this is to distribute the heat dissipation load amongst a greater number of voltage regulators. Each sensor's LED is configured to draw nearly 100mA for maximum brightness, so using a separate regulator for each sensor eliminates the need for the serious heat sinking that would be required if they were regulated together.

IV.7) Embedded Application

Development of the embedded application for the activity monitor required a more serious time investment than I had initially anticipated. This was due in part to my choice to do my development using an unfamiliar development environment (see Appendix D), in part to my limited experience in programming, and also in part to the necessary complexity of the application. A listing of the source code for this application can be found in Appendix C.

Source Code Organization

The source code is a collection of routines and functions which work together to meet the application's requirements. Each function can be categorized according to what requirement it aims to achieve. Functions are organized into a number of files based on their categories, for a total of 9 source files. Communication between functions is performed primarily by means of global variables. Though considered poor practice by many programmers far greater than I, global variables are quite useful for inter-function communication in relatively simple embedded applications. While the content of all source files can be found in Appendix C, I will include a short description of each here:

- button.c – Functions used for debouncing the four pushbuttons involved in the user interface.

- clock.c – Functions used for keeping track of time: the real-time clock.
- dataread.c – Functions used for reading the sensors and storing data.
- interface.c – Functions to manage the user interface.
- lcd.c – Functions to abstract writing to the LCD.
- monitor.c – The top level source file. The main method is contained in this file, and other files are #included here.
- mystring.c – Some simple string utilities to help with the interface.

Organizing the code this way drastically simplified its navigation, and made development a much more efficient process.

Monitoring

The task of monitoring involves reading data from the sensors and storing it into the controller's memory for later retrieval. The sensors will be read by polling their status at a rate of about 1kHz. A beam break will be counted if a sensor transitions from "blocked" to "unblocked." In order to reduce the number of input/output (I/O) pins required to read the 12 sensors, digital multiplexers are used at the inputs and sensors are read in groups of three. Though the sensors are not read at the same time, all of them are read within 25μs. For this application, the sensors are effectively read simultaneously

The 1kHz sampling rate was chosen to balance two potential problems. This rate is sufficiently fast that it will be practically impossible for the monitor to "miss" a beam break, yet it is sufficiently slow that the monitor will remain immune to high speed oscillations around sensor transitions. Also helping to combat this latter problem are schmitt triggers at every input.

As beam breaks are counted, they are written to the appropriate memory location as a large 2 dimensional `unsigned char` array. After each polling interval, the monitor will move on to the next data point, and begin counting anew. For very long polling intervals, it would be possible for there to be more beam breaks than the maximum character (255). The monitor therefore uses a saturation counter, so that if more than 254 breaks are counted, then exactly 254 will be reported.

User Interface

Writing the user interface was the most challenging aspect of this system's implementation, largely due to the difficulty of managing the liquid crystal display (LCD). The microcontroller communicates with the LCD via a 9600 BAUD TTL-level serial connection, which is driven by software. The hardware serial channel is needed for uploading data, and so was not available for communication with the LCD.

To abstract the use of the LCD, I wrote several functions to handle low level tasks. Since the serial link to the LCD is slow, I limited the amount of data that needed to be sent to the LCD by keeping track of "dirty lines," meaning those lines which need to be redrawn.

The interface is implemented with a reasonably large state machine. The four pushbuttons, which are polled every 20ms and debounced, determine the navigation through this state machine. The LCD abstraction routines I wrote make it fairly easy to draw menus, with the selected item in a "reverse video" font (inverted colors), as well as to mange the LCD back-light with a timeout.

IV.8) Final Requirements

- *The system must perform its above described function (it must use infrared beams to measure activity on regular intervals, record its measurements, and make the resulting data available for spreadsheet analysis).*
- *A non-technical person must be able to effectively use the system.*
- *The system must not interfere with the experiment it is intended to measure.*
- *The system must comply with regulations for animal laboratories.*
- *The monitor must be a stand-alone device: no dedicated computer may be used as part of the system.*
- *A computer should not be required to make changes to settings*
- *Monitor status should be available at all times (without computer)*
- *The monitor should be able to store at least 3 days worth of data if a point is taken every 10 minutes*
- *The monitor should support 12 sensors*

V) Results

The activity monitor was fully implemented as described here, and should be entering service soon. Unfortunately, since the monitor has not yet been used for its intended application, there are no sample data to present at the time of this writing. Initial testing indicates that the activity monitor meets all of the requirements which were initially set, and that it should be fairly robust.

V.1) Cost

A cost-effective solution was a significant part of this project's initial motivation, and I believe that this project achieved success in this regard. A brief summary of the costs for this project is listed below.

Description	***Price***
Aluminum enclosure	$10.00
Wire-wrap board	$4.00
Wire-wrap wire	$5.00
Wire-wrap IC Sockets	$30.00
Integrated Circuits (including microcontroller)	$30.00
Serial LCD Display	$100.00
Pushbuttons	$4.00
DB-9 Connector	$2.00
Power Supply	$10.00
Sensor PCBs (incl. Components)	$130.00
Sensor boxes, mounting hardware (Depends on cost of labor)	$200.00
XLR Microphone Cable	$66.00
XLR Connectors	$40.00
Total	**$631.00**

V.2) Generalization

While the activity monitor was designed for use in one specific application, it could easily be adapted for use elsewhere. The sensors simply detect beam breaks, and could be mounted almost anywhere to detect any opaque object. The activity on each

sensor is counted independently, and it is not necessary to use all 12 sensors. The monitor is configurable via the user interface, and so can be adapted to a wide range of applications. If an application requires more in-depth configuration than is possible through the user interface, the microcontroller is in-system programmable and all source code is available.

One of the monitor's features that makes it highly generalizable is the standard interface between the monitor and the sensors. Power is supplied to the sensors, and return a 5 volt logic signal representing whether they are currently “blocked” or “unblocked.” Any sensor which conforms to this interface can be used with the monitor, so long as it will switch slower than the 1kHz rate at which the sensors are checked.

VI) Conclusions

VI.1) Manpower

This project has been an interesting design exercise, particularly because I did all of the work myself. This eliminated any confusion about authority, responsibility, communication, and other problems frequently plaguing group projects, but I believe that the project would have been completed more quickly, more intelligently, and more thoroughly if more people were involved. More minds would have brought more ideas, and more manpower would have brought the ability to work and research in parallel. The implementation was simplified to limit the scope of this project in ways that would not have been necessary if even just one more mind were involved.

VI.2) The Weakest Links

Though the activity monitor does work well, there are some areas of the technical design that I feel are weak. Were I to design a "version 2," these are the areas I would look to improve.

The Atmel 90S8515 microcontroller was chosen as much for its availability and familiarity as it was for its suitability. As I have mentioned, the capabilities of the 8515 are somewhat lean for this application. While this has little impact on the functionality of the final product, it results in a cost in development time and in complexity (i.e. maintainability) of the embedded application. A few specific features of the controller which would be useful in this application are listed below.

- Hardware support for an additional Universal Serial Receive/Transmit (UART)
  to avoid the need to write a software UART
- Large internal memory, perhaps non-volatile memory
  To avoid the additional hardware required for external memory, and possibly render the system immune to power loss
- More I/O port pins
  To avoid the need to conserve them by using multiplexers and the serial LCD display

Another weak area of the activity monitor's design is the sensors. They are acceptable for well controlled environments, but they are also highly susceptible to ambient lighting conditions. As designed, the detector portion of the sensors simply detects the intensity of IR incident on it and the emitter provides a bright IR source. This solution was chosen largely because I did not have the time to implement the solution mentioned in section III.4 above. This alternative which I considered would involve transmitting some carrier on the emitter and selectively sensing that signal at the detector.

In retrospect, the sensors may be the very weakest link of the whole system. This weakness, however, serves to illustrate one of the strengths of the design. Fortunately, the system is designed such that the current sensors could easily be replaced by an entirely different implementation. Since the interface between the sensors and the monitor is simply a 5 volt digital logic signal, virtually any sensor could be made to work with the activity monitor. If, at some later time, it is decided that the original sensors are inadequate, a drop-in replacement could be designed.

VII) Acknowledgments

This project would not have been conceived without the research of Dr. Howland. His input was crucial in the design of this system, as was his patience with its unexpectedly drawn-out completion. Dr. Land provided excellent support and guidance throughout the development effort. Much needed help with the mechanical aspects of this system was provided by Gary Oltz.

Appendix A: Design Proposal

The pages that follow are the design proposal originally drafted for Dr. Howland at the early conceptual stages of this project's design. This document represents many of the design decisions as they were viewed at the onset of the project.

Activity Monitor
Design Options

Derek Brader

Interface

Reporting Method: *constant* vs. *interval*

I would call a constant reporting configuration one in which there is a persistent serial (RS-232) connection between the monitor and the computer to which it is reporting. Trade-offs for this method:

- [pro] constant/immediate availability of data
- [pro] generally immune to microcontroller malfunction
- [pro] (effectively) indefinite reporting without user intervention
- [con] requires dedicated computer near the monitor
- [con] relies on integrity of the RS232 cable
- [con] relies on the stability of the dedicated computer's operating system

An interval reporting configuration is one in which data must be periodically retrieved from the monitor, perhaps via a notebook computer. Trade-offs for this method are essentially the inverse of those listed above:

- [pro] no dedicated computer required
- [pro] computer must only remain stable for a short time
- [con] the periodicity of the data downloads is constrained by the rate at which data accumulates and the size of the microcontroller's memory (-> no long vacations).
- [con] microcontroller malfunction will cause data loss

Software Options: H*yperTerm* vs. *custom application*

It would be possible to download data using the Windows program HyperTerm, or a similar readily-available product. Another option is to develop a custom application to interface with the microcontroller. Trade-offs for using HyperTerm:

- [pro] well tested bug free (hopefully) software
- [pro] familiar interface (maybe)
- [pro] less development time

- [con] HyperTerm may lack necessary features
- [con] HyperTerm's interface may not ideally suit this application

Direct user interface: LCD and Keypad – how much is required?

Rather than interface with the monitor using only a command-line interface on a PC, it may be useful to have an LCD display and small keypad on the monitor to perform basic commands and functions. How much functionality is desired from such an interface must be decided. Clearly, cost will be affected size and features of the display and keypad. There are many things the display and keypad might be used for, but here are some ideas:

- show time/memory remaining
- show current activity
- browse brief activity history
- display or set time/date
- display or set polling interval
- test/calibration modes

Structure

There are many possibilities for sensor mounting structures. I will suggest a couple I have in mind.

Frame mount structure

A frame mount structure could rest on the ground with “arms” coming up to hold the sensors, or it could rest on top of the cage with “arms” reaching down to hold the sensors. This type of structure would be somewhat large, but it would not have to be calibrated frequently because the structure could be easily moved without changing the relative positions of the sensors. The sensor height could also be easily adjusted with this structure.

Modular structure

The idea of this structure is that the sensors would sit on a pedestal (like a cement block) beside the cage. This offers great flexibility in sensor placement and orientation, and also in portability and re-configuration. Since the sensors would be easily moved, however, re-calibration might frequently be necessary.

Some sketches of what I have in mind are on the following page.
[These sketches were hand-drawn on the original document, and will not appear here. The original sketches were not followed.]

Cabling

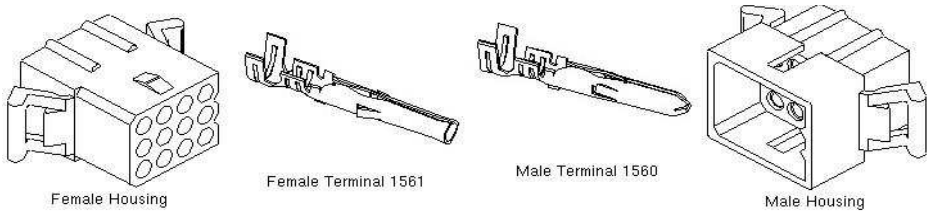
Wires must connect the sensors to the monitor's main circuitry. Possible solutions vary in robustness, ease of use, and price.

Terminal strip: (very low cost)

Probably the least expensive option, wires in twisted pairs could connect terminal strips at the monitor circuitry to those at the sensors. This solution is cost effective and fairly robust, but is by far the most difficult to use or move since each wire must be connected individually to two screw terminals.

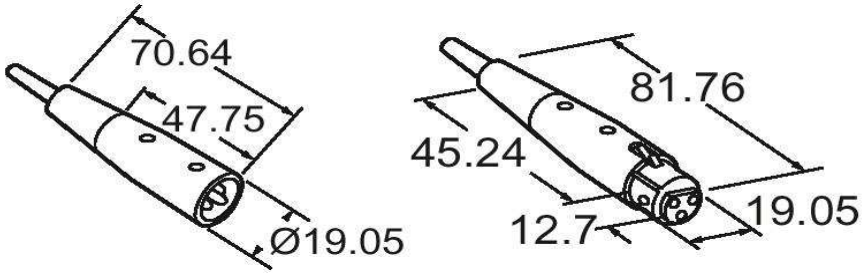
MOLEX connectors: (≈$20 w/o cables)

Inexpensive MOLEX connectors (such as those used in automobile electrical systems and computer power supplies) could be used to simplify wiring. Since many wires can share the same plug, the user is required to make few connections and the risk of making wrong (or poor) connections is minimized. Connectors such as these would also open the possibility of panel mounting receptacles for cleaner wiring.



XLR connectors: (≈$140 w/cables)

For robust noise immunity, triaxial cables may be desired to carry signals to/from the sensors. While other connectors could be used for this type of cable, the best connectors are probably the XLR connectors frequently used with low-impedance microphones. While somewhat expensive, these latching connectors are sufficiently indestructible to survive abuse on stage by rock musicians, yet of high enough quality to be used in fine recording studios. They will also be easy to use, since we will probably require one connector per IR emitter/detector pair. It is also possible to panel-mount these connectors.



Costs

Costs should be fairly low, depending on which connectors are chosen. The parts required for the circuitry and mounting hardware should not exceed $500 unless a large LCD display is desired.

Appendix B: Instructions for Use of the Activity Monitor

The activity monitor uses 880 nanometer (infrared) light beams to detect "activity," which is to be quantified by the number of beam breaks over a given time interval. A single control module with 32kB of memory accumulates data from 12 independent sensors, and is capable of uploading that data over an industry-standard RS-232 serial data link.

Organization

- Monitor Configuration: Using the LCD Interface (Menus and setup)
- Hardware: Setting Up the Sensors (Parts list, hardware setup, system calibration)
- Adjusting Receiver Sensitivity Threshold (Receiver calibration)
- Uploading Data (Interfacing with a computer)

Monitor configuration: Using the LCD Interface

When the control module is plugged in, its LCD will briefly display "Clearing Memory" followed quickly by the main status screen. The main status screen will show the monitor's real-time clock, and it will display "RESET!" at the bottom of the screen to show that the monitor has been reset and is not configured. *NOTE: The monitor will lose all configuration and currently held data if it loses power!* Before use, the monitor should be configured using the LCD and the 4 pushbuttons on the control module. In order for the monitor to behave as expected, the time, date, and polling interval must be set. The hardware also must be properly set up and calibrated.

The four pushbuttons are arranged like arrows on a computer keyboard, and you should think of them as having the following functions:

`right`: select, forward, yes

`left`: un-select, backward, no

`up`: increment

`down`: decrement

From the status screen, push `left`, `up`, or `down` to momentarily activate the back-light of the LCD. Push `right` to activate the main menu (the back-light will remain on while

navigating the menus and configuration). In the main menu, push `right` to select a menu item, `up` or `down` to change the selection, or `back` to exit to the main status screen. Options available in the main menu are “View Current Status,” “Upload Data,” and “Clear Memory.” Selecting “More Options...” from the main menu will display the secondary menu, including the options “Set Time/Date,” “Set Polling Interval,” and “Calibration.” The explanation of these functions, found below, should describe most of the control module's functionality.

View Current Status: This will display the current memory usage as both a percentage of available memory, and as a number of points taken out of the total number of points that can be taken. Push `left` or `right` to return to the main status screen.

Upload Data: Selecting this option will cause the monitor to dump the data it contains to the serial port. See the section on uploading data below.

Clear Memory: Selecting this option will display a screen requesting confirmation to clear the memory. Press `right` to clear the memory, and `left` to cancel. Either choice will bring you back to the main status screen. Clearing the memory will cause all currently stored data to be lost, and previously used memory will be recovered for more logging. Clearing the memory will not affect monitor settings such as the time, date, and polling interval.

Set Time/Date: This option allows you to set the monitor's real-time clock. The time and date will be displayed with a cursor (^^) under one of the fields. Use the `left` and `right` buttons to move the cursor, and the `up` and `down` buttons to adjust the value of the field above the cursor. Moving the cursor off the screen will exit “Set Time/Date” mode and the monitor will ask for confirmation of a memory clear. Press `right` to clear the memory, and `left` to set the clock without clearing the memory. Either selection will bring you back to the main status screen. *NOTE: If the clock is set without clearing the memory, then data points taken prior to setting the clock will have erroneous time stamps when uploaded. Clearing the memory is therefore recommended whenever the clock is set.*

Set Polling Interval: The polling interval is the period at which data points are recorded. The period is expressed in minutes, and the default is 10. A short polling interval

will yield more precise data, but will cause the memory to fill up faster. A long polling interval conserves memory at the expense of time-precision of the data. The amount of time the monitor can log data for without running out of memory can be determined as follows:

$$T = 43.33 \cdot P$$

Where P is the polling interval in minutes and T is the time, in hours, before the memory will run out. Change the value of the polling interval with the `up` and `down` buttons. Press `left` to discard changes to the polling interval and return to the main status screen. Press `right` to accept changes to the polling interval. Accepting changes will cause the monitor to ask for confirmation of a memory clear. Press `right` to clear the memory, and `left` to change the polling interval without clearing the memory. Either selection will bring you back to the main status screen. *NOTE: If the polling interval is set without clearing the memory, then data points taken prior to changing the polling interval will have erroneous time stamps when uploaded. Clearing the memory is therefore recommended whenever the polling interval is changed.*

Calibration: It will be important to ensure proper operation of the sensors by calibrating them. The calibration screen will show a sensor number, and an X or an O corresponding to the sensor's current status. X is displayed when the beam is blocked and O is displayed when it is not. A buzzer will also sound when the beam is blocked. Press up or down to change which sensor is being calibrated. Press left or right to exit calibration mode and return to the main status screen. Also see the section below on setting up the sensors.

Hardware: Setting Up the Sensors

Parts

- 1 x Control Module (Aluminum box including LCD, pushbuttons, DB-9 connector for serial communications, and XLR connectors for communication with sensors)
- 12 x 2-conductor shielded cable with XLR connectors

- 12 x Sensor Unit (polystyrene box with XLR connector, tube-mounted infrared detector, and tube-mounted infrared diode attached to 4' cable)

Each of the 12 sensors includes a tube-loaded IR detector in a rectangular polystyrene box and an IR light emitting diode attached with an approximately 4 foot length of cable. An XLR connector is provided on the rectangular box for connection to the control module. Any of the 12 sensors may be connected to any of the 12 sensor inputs on the control module.

Proper setup of the sensors should be fairly straightforward. The IR light emitting diode (LED) must be carefully aimed at the detector box. The beam-spread of the LED is only a few degrees, so it must be aimed very carefully. The detector is far less directionally sensitive than the LED, as the directionality of the detector is limited only by its tube-loading. The LED should be aimed by rotating its PVC housing and adjusting its height. The detector is to be affixed with VELCRO® fasteners in the necessary location on the panel opposite the LED.

The calibration mode discussed above will most likely be necessary when aiming the sensors. A video camera may also prove useful (the imaging units on many video cameras are sensitive to infrared wavelengths). If you have problems aiming the sensors, try starting at a very short distance. If the detector seems unresponsive to the LED, if you cannot achieve sufficient range, or if the detector is too heavily affected by ambient light, it may be necessary to adjust the sensitivity threshold of the detector. See below for help with doing this.

Adjusting Receiver Sensitivity Threshold

It may become necessary to adjust the detector sensitivity threshold. This is the most challenging aspect of configuring the activity monitor. To do this, you must remove the four screws on the detector's polystyrene box and open it. There is a blue rectangular trim potentiometer mounted on the printed circuit board with a white adjustment knob on one side. The adjustment knob can be rotated with a screwdriver to alter the sensitivity threshold, clockwise to decrease sensitivity. While adjusting

sensitivity, you should hold the blackened PVC shielding tube in front of the detector element as it would be held when the detector box is fully assembled. This may be cumbersome, but is necessary for good results. Sensitivity adjustment should always be done in the same lighting conditions under which the sensor will be used. If the sensor is to be used in multiple lighting conditions, set sensitivity under the brightest condition.

Plug the sensor into the control module, enter calibration mode, and use the `up` and `down` buttons to select the appropriate sensor. The control module should report that the sensor is not blocked. At this point, the detector is receiving ambient light from the room. Without the LED pointing at the detector (and while holding the PVC shielding tube in place), turn the sensitivity down to the point where the control module just barely shows that the sensor is blocked. The sensor should now be properly calibrated.

Uploading Data

The first step in uploading data is to set up the computer. Connect your computer's serial port to the DB-9 connector on the side of the activity monitor. Open a serial communications program and configure it for 9600 baud with no flow control, no parity, 8 data bits, and one stop bit. You will probably also wish to configure it to capture to file.

Once this computer setup is complete, activate the main menu and select “Upload Data.” The upload will start immediately. During the upload, a status bar will show progress. Note that if the memory is full, uploading may take a moment to complete. Once the upload is complete, the monitor will confirm a memory clear. Push `right` to clear the memory, `left` to cancel. Either choice will bring you back to the main status screen.

Note that, on the monitor's LCD, the transfer will appear to happen properly regardless of whether or not the computer is properly configured. Do not confirm the memory clear until you confirm that the data has been properly transferred to the computer. If the transfer was not successful, cancel the memory clear, check your connections and software settings, and try again.

Appendix C: Code

```
/*
 *
 * monitor.c
 * this is the highest level file, which includes what others are
 * necessary.  main() is declared here, as well are the ISRs and
 * initialize().
 *
*/

// automagically includes ioXXX.h
#include <io-avr.h>
// interrupt handling stuff
#include <interrupt.h>
// more interrupt stuff
#include <sig-avr.h>
// timer stuff
#include <timer.h>
#include <stdlib.h>


  // lots o' global vars... declare early.
unsigned char reload;  // reload val for timer 0
char *line1, *line2, *line3, *line4;
unsigned char dirty_lines; // lines which need to be refreshed
unsigned char rv_lines;  // lines to be printed in reverse video
char sclock[21];
unsigned char polling_interval;  // time (min) between writing data to mem
unsigned char buzz; // set to one to make the buzzer ring

// things i split off...
#include "dataread.h"
#include "lcd.c"
#include "mystring.c"
#include "clock.c"
#include "button.c"
#include "interface.c"
#include "dataread.c"


void blink( void );
void initialize( void );

INTERRUPT( SIG_OUTPUT_COMPARE1A );
INTERRUPT( SIG_OVERFLOW0 ); // to manage tasks
INTERRUPT( SIG_INTERRUPT0 ); // for clock (use 32768 osc)
//void delay_ms(unsigned short ms);


// TIMER 0 INTERRUPT FOR TASK MANAGEMENT
// should run about every ms.
INTERRUPT(SIG_OVERFLOW0)
{
        // since the button routine is slow (and since the timer 1 isr
        // is far more time critical), we enable interrupts here.
        sei();
        outp(reload,TCNT0);

          // debounce the buttons
```

```
          // if this isn't IN the isr, then the buttons are fairly
          // non-responsive due to all the slow code in main()
        button_check();

        // handle the buzzing...
        // and YES, i do mean inp(PORTB).  we're just looking at the
        // currentvalue of the port, not trying to read the value of
        // the PIN.
        if ( buzz && (inp(PORTB) & 0x10))
                cbi(PORTB,PB4);
        else
                sbi(PORTB,PB4);

        if (t_dr_sencheck > 0) t_dr_sencheck--; // every 1ms
        if (t_backlight > 0) t_backlight--; // every 1000ms
        if (t_interface > 0) t_interface--; // every 100ms
}


// LONG isr... hopefully it'll work
INTERRUPT( SIG_OUTPUT_COMPARE1A )
{
        // SOFTWARE UA_T for LCD
        lcd_isr();
}


INTERRUPT( SIG_INTERRUPT0 )
{
        sei();
//      clock_isr();
//      this function contains only one line, and we want the isr to return
//      quickly so it's much faster to run it "inline".
        subsecond++;
}


int main(void)
{
        initialize();

        // clear the data arrays.
        // this uses the lcd, so it must happen *after* initialize()
        dr_clearmem();

        while(1)
        {

                if (t_interface == 0) interface();
                lcd_refresh(line1, line2, line3, line4, dirty_lines,
rv_lines);
                dirty_lines = 0;

                if (t_backlight == 0) backlight();
                if (t_dr_sencheck == 0) dr_sencheck(); // every

                clock_checkdate();
                dr_poll();
```

```
        }

}


void initialize(void)
{

        // first and foremost, we want LOTS of memory
        // turn on external memory in the mcucr
        // but we need to turn these on REALLY early for anything to
        // work... so that's taken care of with linker options.
        // check out the Makefile and the linker script, avr85xx.x
        //sbi(MCUCR,SRE);
        //sbi(MCUCR,SRW);

        // for the clock, we enable external interrupt pin 0
        sbi(GIMSK,INT0);
        cbi(DDRD,PD2); // input, no pullup
        cbi(PORTD,PD2);
        sbi(MCUCR,ISC01); // interrupt on rising edge
        sbi(MCUCR,ISC00);

        // set up timer1
        outp(0x00,OCR1AH); // set up OCR1A
        outp(0x34,OCR1AL); // for 9600 BAUD
        //outp(0xd0,OCR1AL); // for 2400 BAUD
        outp(CK8, TCCR1B); // start timer1 prescale=8
        sbi(TCCR1B, CTC1); // enable clear on compare a match
        sbi(TIMSK, OCIE1A);

        // unfortunately, we still need timer0
        // should run about once per ms
        reload = 256-62;
        outp(reload,TCNT0);
        sbi(TIMSK,TOIE0);    // timer0 overflow isr on
        timer0_source(CK64); // prescale=64

        lcd_init();
        button_init();
        dr_init();

        // PORTB4 is the buzzer
        sbi(DDRB,PB4);

        day = 1;
        month = 1;
        polling_interval = 10;

        interface_state = interface_start;

        unsigned long int i;
         // kill time for the lcd to start up
        for(i=0; i<0x000fffff; i++);

        sei();
}
```

```
// the port (and pins) which control the data mux and read data
// check in dr_init() when changing these...
#define dr_muxddr DDRD
#define dr_muxport PORTD
#define dr_muxpin1 PD3
#define dr_muxpin2 PD4
#define dr_dataddr DDRB
#define dr_dataport PORTB
#define dr_datapin PINB
#define dr_dmask1 0x20
#define dr_dmask2 0x40
#define dr_dmask3 0x80

// making this too big made some weird memory things happen...
// like dirty_lines stuck at 0x1f or something.  very strange.
// perhaps my calculations for available memory were not so great.
// oh well.  there is still plenty.
#define dr_size 2600 // amount of space to allocate for each sensor
unsigned char sensor[12][dr_size];
unsigned int dr_index; // used to index the sensorXX arrays
unsigned int dr_senstat; // current status of sensors (little endian bitwise)
unsigned char dr_mincount; // count minutes for polling
char dr_pflag; // flag for dr_poll()
#define t_dr_sencheck_reload 1
unsigned char t_dr_sencheck; // task counter for dr_sencheck()
unsigned char dr_month, dr_day, dr_hour, dr_minute; // when polling started


void dr_sencheck ( void ); // check sensors... does most of the work
void dr_init ( void ); // init routine
void dr_clearmem ( void ); // writes zeros to the data arrays
void dr_poll ( void ); // take care of incrementing dr_index (polling)
unsigned char dr_memused ( void ); // create a string w/ % mem used
void dr_upload ( void ); // upload data to serial port
void ser_putchar ( char ); // write character to serial port (wait for ready)
void ser_puts ( char * c ); // write string to serial port (wait for ready)


/*
 *
 * lcd.c
 *
 * this is a bunch of routines i wrote to run the serial lcd, assuming a
 * certain set of global communication variables is not wrecked up by
 * whatever code this is used with.
 *
*/


   // DO NOT USE the following variables in other code...

char lcd_putchar_bit; // for counting bit for soft ua_t
char lcd_putchar_char; // char to send for soft ua_t


// functions in this file:

void lcd_putchar( char ); // puts a single character to serial ua_t tx
void lcd_puts( char c[]); // puts a char array to serial ua_t (using putchar())
```

```
   // sets up some shit.  run this before expecting anything to work.
void lcd_init(void);
   // MUST be run in the timer 1 isr at PRECISELY the baud rate.
void lcd_isr(void);
   // routine to refresh the display.  dirty_lines uses only the first
   // 4 bits, bits 0-3 representing lines 1-4.  when a bit is one in
   // dirty_lines, it means that line is to be refreshed.  rv works like
   // dirty, but ones in this var tell us to print corresponding line in
   // reverse-video.
void lcd_refresh(char *l1, char *l2, char *l3, char *l4,
                 unsigned char dirty, unsigned char rv);
void lcd_backlight_on ( void );
void lcd_backlight_off ( void );


void lcd_puts(char *c)
{
        int i=0;
        while( c[i] != 0 )
        {
                lcd_putchar( c[i] );
                i++;
        }
}

void lcd_putchar( char c )
{
        lcd_putchar_char = c;
        lcd_putchar_bit = 10;

        // wait until the timer 0 isr finishes sending...
        // somehow calling delay_ms here takes FOREVER
        // kind of ghetto, but polling a flag was not working
        // for some reason or another.
        int i;
        for (i = 1; i<0x0fff; i++);


}

void lcd_isr(void)
{
        if (lcd_putchar_bit > 0)
        {
                lcd_putchar_bit--;

                switch(lcd_putchar_bit)
                {
                        case 9: // start bit
                                sbi(PORTD,PD5);
                                break;
                        case 0: // stop bit
                                cbi(PORTD,PD5);
                                break;
                        default:
                                if ((lcd_putchar_char & 0x01) == 0x01)
                                        cbi(PORTD,PD5);
                                else
                                        sbi(PORTD,PD5);
```

```
                                lcd_putchar_char = lcd_putchar_char >> 1;
                }
        }
}


void lcd_refresh(char *l1, char *l2, char *l3, char *l4,
                unsigned char dirty, unsigned char rv)
{
        // if all lines are dirty, clear the screen
        // (as a sanity check or something)
        if(dirty > 0x0f)
                lcd_putchar('\f');


        if(dirty & 0x01)
        {
                if (rv & 0x01)
                        lcd_putchar(2);
                lcd_putchar(1); // home
                lcd_puts(l1);
                if (rv & 0x01)
                        lcd_putchar(3);
        }

        if(dirty & 0x02)
        {
                if (rv & 0x02)
                        lcd_putchar(2);
                lcd_putchar(1); // home
                lcd_putchar('\r');
                lcd_puts(l2);
                if (rv & 0x02)
                        lcd_putchar(3);
        }

        if(dirty & 0x04)
        {
                if (rv & 0x04)
                        lcd_putchar(2);
                lcd_putchar(1); // home
                lcd_puts("\r\r"); // to line 3
                lcd_puts(l3);
                if (rv & 0x04)
                        lcd_putchar(3);
        }

        if(dirty & 0x08)
        {
                if (rv & 0x08)
                        lcd_putchar(2);
                lcd_putchar(1); // home
                lcd_puts("\r\r\r"); // to line 4
                lcd_puts(l4);
                if (rv & 0x08)
                        lcd_putchar(3);
        }
}
```

```
void lcd_backlight_on ( void )
{
        lcd_putchar(14);
}

void lcd_backlight_off ( void )
{
        lcd_putchar(15);
}


void lcd_init(void)
{
        sbi(DDRD,PD5);
        cbi(PORTD,PD5);
        lcd_putchar_bit = 0;
}



/*
 *
 * mystring.c
 *
 * this file contains a bunch of string functions that i wrote.
 * most of them seem redundant and stupid, but they didn't take
 * very long to do.
 *
*/

const char * nlookup = "0123456789";


void inttostring_1(int n, char string_buffer[2])
{
        string_buffer[0] = nlookup[n];
        string_buffer[1] = 0;
}

void inttostring_2(int n, char string_buffer[3])
{
        string_buffer[0] = nlookup[n/10];
        string_buffer[1] = nlookup[n%10];
        string_buffer[2] = 0;
}

void inttostring_3(int n, char string_buffer[4])
{
        string_buffer[0] = nlookup[n/100];
        string_buffer[1] = nlookup[(n%100)/10];
        string_buffer[2] = nlookup[n%10];
        string_buffer[3] = 0;
}

void inttostring_5(int n, char string_buffer[6])
{
        string_buffer[0] = nlookup[n/10000];
        string_buffer[1] = nlookup[(n%10000)/1000];
```

```
        string_buffer[2] = nlookup[(n%1000)/100];
        string_buffer[3] = nlookup[(n%100)/10];
        string_buffer[4] = nlookup[n%10];
        string_buffer[5] = 0;
}



/*
 *
 * clock.c
 *
 * here are a bunch of functions and variables to handle the time
 * and date.
 *
 *
*/


// timer 1 cycles per second:  this is the number of times that the
// clock_isr() routine will be run per second.  (muy importante
// pour accuracyo)
//#define isr_cps 9615.38461538
#define isr_cps 1


char day, month;
char hour;
char minute;
char second;
unsigned char subsecond;
//double subsecond;


// this handles day/month roll-over (different months -> different
// # of days).  this must run periodically for the days and months
// to work out right.  (i.e. at least once per day)
void clock_checkdate(void);
void clock_isr(void); // goes in the timer 1 isr



// at this point, all the clock isr does is increment subseconds.
// this is necessary since it needs to run FAST for the soft ua_t
// to still run at 9600 BAUD.  the rest of the malarchy is taken
// care of in clock_checkdate()
void clock_isr(void)
{
        subsecond++;
}


// this not only handles dates now, but also seconds minutes and whatever
// else.  this is because the external interrupt isr we're using needs
// to run REALLY quickly because it runs so frequently (32768Hz) that we
// start having problems with the software ua_t.
// At this point, of course, we are no longer running the external interrupt
// at that high speed.  this implementation remains nonetheless.
void clock_checkdate(void)
```

```
{
        if(subsecond >= isr_cps)
        {
        subsecond = subsecond - isr_cps;
        second++;
        if(second == 60)
        {
        second = 0;
        minute++;
        if(minute == 60)
        {
        minute = 0;
        hour++;
        if(hour == 24)
        {
        hour = 0;
        day++;
        } // end if hour
        } // end if minute
        } // end if second
        } // end if subsecond


        switch(month)
        {
                case 9:
                case 4:
                case 6:
                case 11:
                        // 30 day hath september...
                        if(day > 30)
                        {
                                day = 1;
                                month++;
                        }
                        break;
                case 2:
                        // except for february...
                        if(day > 28)
                        {
                                day = 1;
                                month++;
                        }
                        break;
                case 12:
                        if(day > 31)
                        {
                                day = 1;
                                month = 1;
                        }
                default:
                        // all the rest have 31...
                        if(day > 31)
                        {
                                day = 1;
                                month++;
                        }
        }
}
```

```
// this routine generates a pretty string representing the clock.
// the string is exactly 20 characters (plus termination) so that
// it takes up exactly one line of the lcd.
void clock_mkclock(char sclock[21])
{
        char ssec[3], smin[3], shour[3], sday[3], smonth[3];

        inttostring_2(second, ssec);
        inttostring_2(minute, smin);
        inttostring_2(hour, shour);
        inttostring_2(day, sday);
        inttostring_2(month, smonth);

        sclock[0] = shour[0];
        sclock[1] = shour[1];
        sclock[2] = ':';
        sclock[3] = smin[0];
        sclock[4] = smin[1];
        sclock[5] = ':';
        sclock[6] = ssec[0];
        sclock[7] = ssec[1];
        sclock[8] = ' ';
        sclock[9] = ' ';
        sclock[10] = ' ';
        sclock[11] = ' ';
        sclock[12] = ' ';
        sclock[13] = ' ';
        sclock[14] = ' ';
        sclock[15] = smonth[0];
        sclock[16] = smonth[1];
        sclock[17] = '/';
        sclock[18] = sday[0];
        sclock[19] = sday[1];
        sclock[20] = 0;
}



/*
 *
 * button.c
 *
 * set of functions for debouncing 4 pushbuttons.  interface with this code
 * using the global variables defined here.  each is incremented when it is
 * pressed, or when key repeat occurs.
 *
*/

// states for debouncing
#define nopush 0
#define maybepush 1
#define push 2
#define maybenopush 3

// number of times button_check runs before actually doing anything.
// use this to control the debounce time base.
#define button_treload 20
```

```
#define button_repeatdly 500
#define button_repeatpd 180
// checkout button_init if you want to change the pins used...
#define button_port PORTB
#define button_pin PINB
#define button_ddr DDRB


// INTERFACE GLOBALS
// these guys get incremented as buttons are pushed, and should be
// decremented by accompanying code as actions are taken.
unsigned char button_select;
unsigned char button_back;
unsigned char button_plus;
unsigned char button_minus;

// other button vars...
unsigned char button_state;     // used in button_check() to keep state
unsigned char button_t;         // timebase in button_check()
unsigned char button_repeat;    // counter for key repeat


void button_check(void);
void button_init(void);


void button_check(void)
{
        // do nothing unless we have counted to button_treload
        if(button_t-- != 0)
                return;

        unsigned char button_bread; // used to read input port
        button_t = button_treload;

        button_bread = ~inp(button_pin) & 0x0f;
        // unfortunately a couple of our pushbuttons are normally
        // closed.  just invert those so we don't have to think
        // about it.
        button_bread = button_bread ^ 0x0c;

        switch (button_state)
        {
                case nopush:
                        if (button_bread != 0)
                                button_state = maybepush;
                        break;

                case maybepush:
                        if (button_bread != 0)
                        {
                                button_state = push;
                                button_repeat = button_repeatdly /
button_treload;
                                if (button_bread == 1)
                                        button_minus++;
                                if (button_bread == 2)
                                        button_plus++;
                                if (button_bread == 4)
                                        button_back++;
```

```
                                if (button_bread == 8)
                                        button_select++;
                        }
                        else
                                button_state = nopush;
                        break;

                case push:
                        if (button_bread == 0)
                                button_state = maybenopush;
                        else if ( button_repeat-- == 0 )
                        {
                                button_repeat = button_repeatpd /
button_treload;
                                if (button_bread == 1)
                                        button_minus++;
                                if (button_bread == 2)
                                        button_plus++;
                                if (button_bread == 4)
                                        button_back++;
                                if (button_bread == 8)
                                        button_select++;
                        }

                        break;

                case maybenopush:
                        if (button_bread == 0)
                                button_state = nopush;
                        else
                                button_state = push;
        }
}


void button_init(void)
{
        sbi(button_ddr,0);
        sbi(button_ddr,1);
        sbi(button_ddr,2);
        sbi(button_ddr,3);
        sbi(button_port,0);
        sbi(button_port,1);
        sbi(button_port,2);
        sbi(button_port,3);
        button_t = button_treload;
        button_state = nopush;
}


/*
 *
 * tasks.c
 *
 * functions here are tasks to be run and managed by code elsewhere.
 * this is mostly just for the sake of organization, and to keep the
 * files a litte smaller.
 *
```

```
*/

   // task counter for backlight() (should run about every sec)
#define t_backlight_reload 1000
#define backlight_timeout 5 // how long before we kill the b-light
unsigned short t_backlight;
   // number of seconds (depends on t_backlight_reload) till b-light killed
char backlight_count;
char smemused[21];

   // the interface state machine.  this will handle the button/lcd user
   // interface.
#define t_interface_reload 100
#define interface_start 0
#define interface_status 1
#define interface_mainmenu 2
#define interface_settime 3
#define interface_setpi 4
#define interface_menu2 5
#define interface_clearconfirm 6
#define interface_stat1 7
#define interface_stat2 8
#define interface_calibrate 9
unsigned char t_interface;
unsigned char interface_state;
unsigned char interface_sel;


void interface ( void );        // state machine for user interface
void mainmenu (char sel);       // displays the main menu
void menu2 ( char sel );        // displays the second menu
void tostate_status ( void );   // takes machine to status state
void tostate_clearconfirm ( void ); // takes machine to clearconfirm state
void backlight( void );         // run as task.  kills backlight after timeout


char spolling_interval[4];
unsigned char newpolling_interval;
unsigned char interface_sensor = 0; // for sensor selection during calibration



// handle the backlight timeout.
// this should be scheduled as a task in main.
void backlight( void )
{
        t_backlight = t_backlight_reload;
        if ( backlight_count == 0 )
                lcd_backlight_off();
        else
                backlight_count--;
}


// The interface state machine.  this is the bulk of this file,
// and is also the bulk of the user interface.  i feel like i
// should have more to say about it here at the top, but it's
// about as straightforward as a case statement this size can be.
void interface ( void )
{
```

```
t_interface = t_interface_reload;
unsigned short i;

switch(interface_state)
{
        case interface_start:
                interface_sel = 0;
                tostate_status();
                line3 = "RESET!!!";

                break;

        case interface_status:
                if(button_back || button_plus || button_minus)
                {
                        lcd_backlight_on();
                        backlight_count = backlight_timeout;
                        button_back = 0;
                        button_plus = 0;
                        button_minus = 0;
                }

                if( button_select )
                {
                        lcd_backlight_on();
                        backlight_count = backlight_timeout;
                        button_select--;
                        interface_sel = 0;
                        mainmenu(interface_sel);
                        dirty_lines = 0x1f;
                        interface_state = interface_mainmenu;
                }
                else
                {
                        clock_mkclock(sclock);
                        line2 = sclock;
                        dirty_lines = dirty_lines | 0x02;
                }


                break;

        case interface_mainmenu:
                backlight_count = backlight_timeout;

                if (button_plus)
                {
                        interface_sel = (interface_sel + 3) % 4;
                        mainmenu(interface_sel);
                        button_plus--;
                }
                if (button_minus)
                {
                        interface_sel = (interface_sel + 1) % 4;
                        mainmenu(interface_sel);
                        button_minus--;
                }

                if( button_back )
                {
```

```
                        button_back--;
                        tostate_status();
                }
                if ( button_select )
                {
                        button_select = 0;
                        if ( interface_sel == 0 ) // "more options"
selected
                        {
                                menu2(interface_sel);
                                dirty_lines = 0x1f;
                                interface_state = interface_menu2;
                        }
                        if ( interface_sel == 1 ) // "view status"
selected
                        {
                                line1 = "";
                                line2 = "";
                                line3 = "";
                                line4 = "";
                                dirty_lines = 0x1f;
                                rv_lines = 0x00;
                                interface_state = interface_stat1;
                        }
                        if ( interface_sel == 2 ) // "upload" selected
                        {
                                dr_upload();
                                button_back = 0;
                                button_select = 0;
                                button_plus = 0;
                                button_minus = 0;
                                tostate_clearconfirm();
                        }
                        if ( interface_sel == 3 ) // "clear memory"
selected
                        {
                                tostate_clearconfirm();
                        }

                }

                break;

        case interface_menu2:
                backlight_count = backlight_timeout;

                if (button_plus)
                {
                        interface_sel = (interface_sel + 2) % 3;
                        menu2(interface_sel);
                        button_plus--;
                }
                if (button_minus)
                {
                        interface_sel = (interface_sel + 1) % 3;
                        menu2(interface_sel);
                        button_minus--;
                }

                if( button_back )
```

```
                        {
                                button_back--;
                                tostate_status();
                        }
                        if ( button_select )
                        {
                                button_select = 0;
                                if ( interface_sel == 0 ) // "set time/date"
selected
                                {
                                        clock_mkclock(sclock);
                                        line1 = "   Set Time/Date";
                                        line2 = "";
                                        line3 = sclock;
                                        line4 = "^^";
                                        dirty_lines = 0x1f;
                                        rv_lines = 0;
                                        interface_state = interface_settime;
                                        interface_sel = 0;
                                }
                                if ( interface_sel == 1 ) // "set polling
interval" selected
                                {
                                        newpolling_interval =
polling_interval;
                                        inttostring_3(newpolling_interval,
spolling_interval);
                                        line1 = "Set Polling Interval";
                                        line2 = "     (minutes)";
                                        line3 = spolling_interval;
                                        line4 = "^^^";
                                        dirty_lines = 0x1f;
                                        rv_lines = 0;
                                        interface_state = interface_setpi;
                                        interface_sel = 0;
                                }
                                if ( interface_sel == 2 ) // "Calibrate"
selected
                                {
                                        line1 = "\tCalibrating";
                                        // no, we don't set interface_sensor
to zero.
                                        // this is a bug i've not yet found...
regardless
                                        // of what we put here, the previous
sensor # will
                                        // appear on the screen...
                                        line2 = "\t Sensor 01";
                                        line3 = "";
                                        line4 = "";
                                        rv_lines = 0x00;
                                        dirty_lines = 0x1f;
                                        interface_state = interface_calibrate;
                                }
                        }

                        break;

                case interface_settime:
                        clock_mkclock(sclock);
```

```
                        line3 = sclock;
                        dirty_lines = dirty_lines | 0x04;
                        backlight_count = backlight_timeout;
                        switch(interface_sel)
                        {
                                case 0: // adjusting hours
                                        if ( button_plus )
                                        {
                                                hour++;
                                                if ( hour > 23 )
                                                        hour = 0;
                                                button_plus--;
                                        }
                                        if ( button_minus )
                                        {
                                                hour--;
                                                if ( hour < 0 )
                                                        hour = 23;
                                                button_minus--;
                                        }
                                        if ( button_select )
                                        {
                                                line4 = "   ^^";
                                                dirty_lines = dirty_lines |
0x08;
                                                interface_sel++;
                                                button_select--;
                                        }
                                        if ( button_back )
                                        {
                                                tostate_clearconfirm();
                                                button_back--;
                                        }
                                        break;

                                case 1: // adjusting minutes
                                        if ( button_plus )
                                        {
                                                minute++;
                                                if ( minute > 59 )
                                                        minute = 0;
                                                button_plus--;
                                        }
                                        if ( button_minus )
                                        {
                                                minute--;
                                                if ( minute < 0 )
                                                        minute = 59;
                                                button_minus--;
                                        }
                                        if ( button_select )
                                        {
                                                line4 = "      ^^";
                                                dirty_lines = dirty_lines |
0x08;
                                                interface_sel++;
                                                button_select--;
                                        }
                                        if ( button_back )
                                        {
```

```
                                        line4 = "^^   ";
                                        dirty_lines = dirty_lines |
0x08;
                                        interface_sel--;
                                        button_back--;
                                }
                                break;

                        case 2: // adjusting seconds (clear only)
                                if ( button_plus | button_minus )
                                {
                                        second = 0;
                                        subsecond = 0;
                                        button_plus = 0;
                                        button_minus = 0;
                                }
                                if ( button_select )
                                {
                                        line4 = "               ^^";
                                        dirty_lines = dirty_lines |
0x08;
                                        interface_sel++;
                                        button_select--;
                                }
                                if ( button_back )
                                {
                                        line4 = "   ^^   ";
                                        dirty_lines = dirty_lines |
0x08;
                                        interface_sel--;
                                        button_back--;
                                }
                                break;

                        case 3: // adjusting months
                                if ( button_plus )
                                {
                                        month++;
                                        if (month > 12)
                                                month = 1;
                                        button_plus--;
                                }
                                if ( button_minus )
                                {
                                        month--;
                                        if ( month < 1 )
                                                month = 12;
                                        button_minus--;
                                }
                                if ( button_select )
                                { // on to days
                                        line4 = "                  ^^";
                                        dirty_lines = dirty_lines |
0x08;
                                        interface_sel++;
                                        button_select--;
                                }
                                if ( button_back )
                                { // back to seconds
                                        line4 = "      ^^        ";
```

```
                                        dirty_lines = dirty_lines |
0x08;
                                        interface_sel--;
                                        button_back--;
                                }
                                break;

                        case 4: // adjusting days
                                if ( button_plus )
                                {
                                        day++;
                                        button_plus--;
                                }
                                if ( button_minus )
                                {
                                        day--;
                                        if ( day < 1 )
                                                day = 28; // this is
dumb
                                        button_minus--;
                                }
                                if ( button_select )
                                { // back to status
                                        button_select--;
                                        tostate_clearconfirm();
                                }
                                if ( button_back )
                                { // back to months
                                        line4 = "               ^^   ";
                                        dirty_lines = dirty_lines |
0x08;
                                        interface_sel--;
                                        button_back--;
                                }
                                break;
                        }  // end switch ( interface_sel )
                        break;  // end of case interface_settime:

                case interface_setpi:
                        backlight_count = backlight_timeout;
                        inttostring_3(newpolling_interval, spolling_interval);
                        line3 = spolling_interval;
                        dirty_lines = 0x04;

                        if ( button_plus )
                        {
                                newpolling_interval++;
                                button_plus--;
                        }
                        if ( button_minus )
                        {
                                newpolling_interval--;
                                button_minus--;
                        }
                        if ( button_back )
                        {
                                button_back = 0;
                                tostate_status();
                        }
                        if ( button_select )
```

```
                {
                        button_select = 0;
                        polling_interval = newpolling_interval;
                        tostate_clearconfirm();
                }
                break; // end case interface_setpi:

        case interface_clearconfirm:
                backlight_count = backlight_timeout;

                if ( button_plus )
                        button_plus = 0;
                if ( button_minus )
                        button_minus = 0;
                if ( button_back )
                {
                        button_back = 0;
                        tostate_status();
                }
                if ( button_select )
                {
                        button_select = 0;
                        dr_clearmem();
                        tostate_status();
                }
                break;

        case interface_stat1:
                backlight_count = backlight_timeout;
                unsigned char memused;
                char smemused[6];
                memused = dr_memused();
                inttostring_3( memused,smemused );
                line1 = "Mem Used:    %";
                for (i=0; i<3; i++)
                        line1[i+10] = smemused[i];


                // another attempted hash function ( hopefully works )
                // use graphics to draw a nice thick line where lines
3&4 would be
                for(i=0; i<10; i++)
                {
                        //lcd_putchar(27); // escape
                        //lcd_putchar('L'); // draw line
                        //lcd_putchar(64 + 0); // x1
                        lcd_puts("\033L\100");
                        lcd_putchar(64 + 19 + i); // y1
                        lcd_putchar(64 + ((long)dr_index*120)/dr_size);
// x2
                        lcd_putchar(64 + 19 + i); // y2
                }


                line2 = "     /      points";
                inttostring_5(dr_index,smemused);
                for(i=0; i<5; i++)
                        line2[i] = smemused[i];

                inttostring_5(dr_size,smemused);
```

```
        for(i=0; i<5; i++)
                line2[i+6] = smemused[i];


        dirty_lines = 0x07;


        if ( button_plus )
                button_plus = 0;
        if ( button_minus )
                button_minus = 0;
        if ( button_back | button_select )
        {
                button_back = 0;
                button_select = 0;
                tostate_status();
        }
        //if ( button_select ) // on to stat2
        //{
        //      button_select = 0;
        //      dirty_lines = 0x1f;
        //      interface_state = interface_stat2;
        //}
        break;
// this doesn't work.
// so we'll forget about it
/*
case interface_stat2:
        backlight_count = backlight_timeout;
        char tmp[12][4];

        line1 = "Current count:";
        line2 = "                    ";
        line3 = "                    ";
        line4 = "                    ";

        for(i=0; i<12; i++)
                inttostring_3( sensor[i][dr_index],tmp[i] );

        for (i=0; i<3; i++)
        {
                line2[i+1] = tmp[0][i];
                line2[i+6] = tmp[1][i];
                line2[i+11] = tmp[2][i];
                line2[i+16] = tmp[3][i];
        }
        for (i=0; i<3; i++)
        {
                line3[i+1] = tmp[4][i];
                line3[i+6] = tmp[5][i];
                line3[i+11] = tmp[6][i];
                line3[i+16] = tmp[7][i];
        }
        for (i=0; i<3; i++)
        {
                line4[i+1] = tmp[8][i];
                line4[i+6] = tmp[9][i];
                line4[i+11] = tmp[10][i];
                line4[i+16] = tmp[11][i];
        }
```

```
                        dirty_lines = 0x0f;


                        if ( button_plus )
                                button_plus = 0;
                        if ( button_minus )
                                button_minus = 0;
                        if ( button_back ) // back to stat 1
                        {
                                button_back = 0;
                                line1 = "";
                                line2 = "";
                                line3 = "";
                                line4 = "";
                                dirty_lines = 0x1f;
                                interface_state = interface_stat1;
                        }
                        if ( button_select )
                        {
                                button_select = 0;
                                tostate_status();
                        }

                        break;
                */

                case interface_calibrate:
                        backlight_count = backlight_timeout;

                        if ( button_plus )
                        {
                                button_plus--;
                                interface_sensor = (interface_sensor + 1) % 12;
                                dirty_lines = dirty_lines | 0x02;
                        }
                        if ( button_minus )
                        {
                                button_minus--;
                                interface_sensor = (interface_sensor + 11) %
12;
                                dirty_lines = dirty_lines | 0x02;
                        }

                        unsigned int mask = 0x0001 << interface_sensor;
                        char interface_ssensor[3];
                        inttostring_2(interface_sensor+1,interface_ssensor);
                        for(i=0; i<2; i++)
                                line2[i+9] = interface_ssensor[i];

                        if ( dr_senstat & mask )
                        {
                                line3 = "\t\t X";
                                buzz = 1;
                        }
                        else
                        {
                                line3 = "\t\t O";
                                buzz = 0;
                        }
                        dirty_lines = dirty_lines | 0x04;
```

```
                        if ( button_select | button_back )
                        {
                                button_select = 0;
                                button_back = 0;
                                buzz = 0;
                                tostate_status();
                        }

                        break;

                default: // we should never get here.
                        line1 = "ERROR";
                        dirty_lines = 0x01;
        }  // end switch ( interface_state )
}


void mainmenu ( char sel )
{
        line1 = "More Options...     ";
        line2 = "View Current Status ";
        line3 = "Upload Data         ";
        line4 = "Clear Memory        ";
        rv_lines = 1 << sel;
        dirty_lines = dirty_lines | 0x0f;
}


void menu2 ( char sel )
{
        line1 = "Set Time/Date       ";
        line2 = "Set Polling Interval";
        line3 = "Calibration";
        line4 = "";
        rv_lines = 1 << sel;
        dirty_lines = dirty_lines | 0x0f;
}


void tostate_status ( void )
{
        clock_mkclock(sclock);
        line1 = "  ACTIVITY MONITOR";
        line2 = sclock;
        line3 = "";
        line4 = "";
        dirty_lines = 0x1f;
        rv_lines = 0;
        interface_state = interface_status;
}


void tostate_clearconfirm ( void )
{
        line1 = "    Clear memory";
        line2 = "";
        line3 = "      Proceed?";
        line4 = "";
        dirty_lines = 0x1f;
```

```
        rv_lines = 0x00;
        interface_state = interface_clearconfirm;
}


/*
 *
 * dataread.c
 *
 * this file houses the routines that handle the gathering of data.  This
 * includes setting outputs to drive the necessary multiplexer, as well as
 * storing the data in large arrays (external ram required).
 *
*/


// ALL DECLARATIONS ARE IN dataread.h

// This routine will drive the muxes, read the sensors, write to memory,
// and take care of dr_senstat.  few functions beyond this should be
// required in this file. (famous last words...)
// unfortunately, due to the number of ports available, we must read the
// sensors in four sets of three (and use multiplexers).
void dr_sencheck ( void )
{
        t_dr_sencheck = t_dr_sencheck_reload;
        unsigned char tmp_read;
        unsigned short int i;


        // ******************
        // PASS 1: SENSORS 1-3
        tmp_read = inp(dr_datapin);
        // set mux for next read "early"
        sbi(dr_muxport, dr_muxpin1);

        if ( tmp_read & dr_dmask1 )
        {
                  // only increment if sensor was previously 0
                if ( (dr_senstat & 0x0001) == 0 )
                        ++sensor[0][dr_index];

                dr_senstat = (dr_senstat | 0x0001);
        }
        else
        {
                  // clear dr_senstat entry
                dr_senstat = dr_senstat & 0xfffe;
        }

        if ( tmp_read & dr_dmask2 )
        {
                if ( (dr_senstat & 0x0002) == 0 )
                        ++sensor[1][dr_index];
                dr_senstat = dr_senstat | 0x0002;
        }
        else
        {
                dr_senstat = dr_senstat & 0xfffd;
```

```
}

if ( tmp_read & dr_dmask3 )
{
        if ( (dr_senstat & 0x0004) == 0 )
                ++sensor[2][dr_index];
        dr_senstat = dr_senstat | 0x0004;
}
else
{
        dr_senstat = dr_senstat & 0xfffb;
}

// ******************
// PASS 2: SENSORS 4-6

tmp_read = inp(dr_datapin);
// set mux for next read
cbi(dr_muxport, dr_muxpin1);
sbi(dr_muxport, dr_muxpin2);
if ( tmp_read & dr_dmask1 )
{
        if ( (dr_senstat & 0x0008) == 0 )
                ++sensor[3][dr_index];

        dr_senstat = dr_senstat | 0x0008;
}
else
{
        dr_senstat = dr_senstat & 0xfff7;
}

if ( tmp_read & dr_dmask2 )
{
        if ( (dr_senstat & 0x0010) == 0 )
                ++sensor[4][dr_index];
        dr_senstat = dr_senstat | 0x0010;
}
else
{
        dr_senstat = dr_senstat & 0xffef;
}

if ( tmp_read & dr_dmask3 )
{
        if ( (dr_senstat & 0x0020) == 0 )
                ++sensor[5][dr_index];
        dr_senstat = dr_senstat | 0x0020;
}
else
{
        dr_senstat = dr_senstat & 0xffdf;
}


// ******************
// PASS 3: SENSORS 7-9

tmp_read = inp(dr_datapin);
// set mux for next read
```

```
sbi(dr_muxport, dr_muxpin1);
if ( tmp_read & dr_dmask1 )
{
        if ( (dr_senstat & 0x0040) == 0 )
                ++sensor[6][dr_index];

        dr_senstat = dr_senstat | 0x0040;
}
else
{
        dr_senstat = dr_senstat & 0xffbf;
}

if ( tmp_read & dr_dmask2 )
{
        if ( (dr_senstat & 0x0080) == 0 )
                ++sensor[7][dr_index];
        dr_senstat = dr_senstat | 0x0080;
}
else
{
        dr_senstat = dr_senstat & 0xff7f;
}

if ( tmp_read & dr_dmask3 )
{
        if ( (dr_senstat & 0x0100) == 0 )
                ++sensor[8][dr_index];
        dr_senstat = dr_senstat | 0x0100;
}
else
{
        dr_senstat = dr_senstat & 0xfeff;
}


// *********************
// PASS 4: SENSORS 10-12

tmp_read = inp(dr_datapin);
// set mux for next (first) read
cbi(dr_muxport, dr_muxpin1);
cbi(dr_muxport, dr_muxpin2);
if ( tmp_read & dr_dmask1 )
{
        if ( (dr_senstat & 0x0200) == 0 )
                ++sensor[9][dr_index];

        dr_senstat = dr_senstat | 0x0200;
}
else
{
        dr_senstat = dr_senstat & 0xfdff;
}

if ( tmp_read & dr_dmask2 )
{
        if ( (dr_senstat & 0x0400) == 0 )
                ++sensor[10][dr_index];
        dr_senstat = dr_senstat | 0x0400;
```

```
        }
        else
        {
                dr_senstat = dr_senstat & 0xfbff;
        }

        if ( tmp_read & dr_dmask3 )
        {
                if ( (dr_senstat & 0x0800) == 0 )
                        ++sensor[11][dr_index];
                dr_senstat = dr_senstat | 0x0800;
        }
        else
        {
                dr_senstat = dr_senstat & 0xf7ff;
        }


        // ****************************
        // ensure saturation of counters
        for(i=0; i<12; i++)
        {
                if ( sensor[i][dr_index] == 255 )
                        sensor[i][dr_index] = 254;
        }

}



// this routine will increment dr_index every polling_interval.  when
// the memory is full, dr_index will no longer be incremented.
// this routine will also print messages to the status screen when
// the memory is low or empty.
void dr_poll ( void )
{
        if ( second == 0 )
        {
                if ( dr_pflag == 0 )
                {
                        dr_mincount++;
                        dr_pflag = 1;
                        if ( dr_mincount == polling_interval )
                        {
                                dr_mincount = 0;
                                if ( dr_index < dr_size-1 )
                                        dr_index++;
                        }
                }
        } else {
                dr_pflag = 0;
        }

        // print warning messages to the status screen...
        if ( interface_state == interface_status )
        {
                if ( dr_index == dr_size - 1 )
                {
                        line3 = "MEMORY FULL!!!    ";
                        dirty_lines = dirty_lines | 0x04;
```

```
                } else if ( dr_index >= ((dr_size*9) / 10) )
                {
                        line3 = "LOW MEM: <10% rem";
                        dirty_lines = dirty_lines | 0x04;
                }
        }
}


// routine to clear the memory arrays (write zeros everywhere)
// This routine uses the lcd, and will take a while to
// complete.  hopefully it won't be toooooo slow.
void dr_clearmem ( void )
{
        lcd_putchar('\f');
        lcd_puts("\n  Clearing Memory");
        lcd_putchar('\r');
        unsigned int i;
        unsigned short int j;
        for (i=0; i<dr_size; i++)
        {
                if ( (i % (dr_size/20)) == 0 )
                        lcd_putchar('#');

                for(j=0; j<12; j++)
                        sensor[j][i] = 0;
        }

        dr_index = 0; // reset index
        dr_mincount = 0; // start a new poll interval now.
          // remember when we start polling.
          // NOTE: if this happens at 11:59pm, we'll remember a day off.
          // but this is sufficiently unlikely that i won't bother
          // checking for it.
        dr_month = month;
        dr_day = day;
        dr_hour = hour;
        dr_minute = minute+1;
        if ( dr_minute == 60)
        {
                dr_minute = 0;
                dr_hour++;
        }
}


// returns the amount of memory used, in percent.
unsigned char dr_memused ( void )
{
        return ((long)dr_index * 100) / (dr_size-1);
}


// This routine uploads data using the hardware UART.  It will
// use the LCD, and will return after data is uploaded.
void dr_upload ( void )
{
        unsigned char u_month = dr_month;
        unsigned char u_day = dr_day;
        unsigned char u_hour = dr_hour;
```

```
unsigned char u_minute = dr_minute;
char u_smonth[3], u_sday[3], u_shour[3], u_sminute[3];
unsigned int i,j;
char u_string[4];

lcd_putchar('\f');
lcd_puts("Uploading...");
lcd_putchar('\r');

unsigned char prog = 1; // for progress

for (i=0; i<=dr_index; i++)
{
          // show progress
        if ( i >= (prog*dr_index/20))
        {
                lcd_putchar('#');
                prog++;
        }

        // exit if cancelled
        if ( button_back )
                return;

        inttostring_2(u_month,u_smonth);
        inttostring_2(u_day,u_sday);
        inttostring_2(u_hour,u_shour);
        inttostring_2(u_minute,u_sminute);

        ser_puts(u_smonth);
        ser_putchar(':');
        ser_puts(u_sday);
        ser_putchar(':');
        ser_puts(u_shour);
        ser_putchar(':');
        ser_puts(u_sminute);

        for(j=0; j<12; j++)
        {
                ser_putchar(',');
                inttostring_3(sensor[j][i], u_string);
                ser_puts(u_string);
        }

        ser_puts("\n\r");

        // add the polling interval to the current date/time
        // (this sucks)
        u_minute = u_minute + polling_interval;
        if(u_minute >= 60)
        {
        u_minute = u_minute - 60;
        u_hour++;
        if(u_hour == 24)
        {
        u_hour = 0;
        u_day++;
        } // end if hour
        } // end if minute
        switch(u_month)
```

```
                {
                case 9:
                case 4:
                case 6:
                case 11:
                        // 30 day hath september...
                        if(u_day > 30)
                        {
                                u_day = 1;
                                u_month++;
                        }
                        break;
                case 2:
                        // except for february...
                        if(u_day > 28)
                        {
                                u_day = 1;
                                u_month++;
                        }
                        break;
                case 12:
                        if(u_day > 31)
                        {
                                u_day = 1;
                                u_month = 1;
                        }
                default:
                        // all the rest have 31...
                        if(u_day > 31)
                        {
                                u_day = 1;
                                u_month++;
                        }
                }
        }
}


// this function will write a string to the serial port.  it will
// not return until is has completed, and note that it requires a
// zero-terminated string.
// this is its own function mostly just for the sake of convenience.
void ser_puts(char * c)
{
        unsigned short int i = 0;

        while ( c[i] != 0 )
        {
                  // wait until the UART is ready to transmit
                while ( (inp(USR) & (0x01 << UDRE)) == 0 ) ;

                outp(c[i],UDR);
                i++;
        }
}


void ser_putchar(char c)
{
          // wait until the UART is ready to transmit
```

```
        while ( (inp(USR) & (0x01 << UDRE)) == 0 ) ;

        outp(c,UDR);
}



// init routine for the dataread functions.
// sets up the ports and a couple varialbles
void dr_init ( void )
{
        sbi(dr_muxddr, dr_muxpin1);
        sbi(dr_muxddr, dr_muxpin2);
        cbi(dr_dataddr, PB5);
        cbi(dr_dataddr, PB6);
        cbi(dr_dataddr, PB7);
        dr_index = 0;
        dr_mincount = 0;
        dr_senstat = 0;
        outp(0x0c,UBRR); // transmit@19200 BAUD
        outp(0x19,UBRR); // transmit@19200 BAUD
        sbi(UCR,TXEN);
}
```

Appendix D: A Linux Development Environment for AVR Microcontrollers using the GNU Compiler Collection

While I had worked with AVR microcontrollers prior to beginning this project, I had only used the Codevision AVR C compiler. I chose to develop the activity monitor on a machine running the Linux operating system. Getting a functional development environment up and running was not incredibly complicated, but I had difficulty finding any consolidated documentation on what was required. I will therefore include here a brief discussion of what I did on my system running Mandrake Linux. This may serve as a useful starting point for future development efforts. Four different packages are required for a functional development environment.

The GNU Compiler Collection (GCC) can compile C source to an AVR target, but GCC itself must be compiled to support this. The source for GCC (a package called gcc-core) can be downloaded from http://gcc.gnu.org. To compile GCC for an AVR target, I ran the configure script as follows:

```
./configure --target=avr --prefix=/usr/local/avr --enable-languages=c –disable-nls
```

Other options should be detected properly by default.

The C libraries will almost surely be necessary for any program you wish to write. A group has developed an AVR C runtime library. This package can be downloaded at http://savannah.nongnu.org/projects/avr-libc/. Scripts are included with this package which make it quite easy to install.

Binutils is a collection of programs for manipulating binaries, including a linker, assembler, and some other tools. These tools are required to generate the hex file which can be programmed onto the target AVR chip. They can be downloaded from http://sources.redhat.com/binutils/. On my system, I configured binutils as follows:

```
./configure --target=avr --prefix=/usr/local/avr
```

The final tool required is the programmer. This application reads the hex file and programs it onto the target AVR using the specified port and programming method. The programmer I used is called uisp (micro in-system programmer) which is available at http://savannah.nongnu.org/projects/uisp/. No configuration was necessary for this program apart from setting the install directory in the Makefile.

In order to program the chip, a programming dongle will be required. Dongles for some programming methods and specific development boards are available

commercially. If one of these is not available, a programming dongle can be made at minimal cost. For the Direct AVR Parallel Access programming method used by UISP, the following connections must be made from the serial port to the target AVR:

Parallel Port	***Target AVR***
Init (pin 16)	Reset
D0 (pin 2)	MOSI (Port B3 for 8515)
Busy (pin 11)	MISO (Port B4 for 8515)
Strobe (pin 1)	SCK (Port B5 for 8515)
Ground (pin 18)	Ground

I made a dongle according to these connections and, though I had some problems with reliability, it did work.

After compiling and installing the four programs discussed above and making the dongle, I was able to program the controller with some simple programs. The final battle with my development environment was getting the external memory to work properly. Two things are required to get external RAM to work. First, the MCU general control register (MCUCR) must be set properly (0x80 for the 8515). Second, the linker must know how much memory is available and where it is.

Both of these tasks must be accomplished by the linker. It is not sufficient to set the MCUCR in user-written initialization code, because once execution jumps to `main()` it is too late. Setting the MCUCR can be achieved by passing the arguments

```
--defsym __init_mcucr__=0x80
```

This will ensure that the linker takes care of the MCUCR.

To give the linker information about the external memory, I used a custom linker script. The linker can be instructed to use the script with the arguments

```
--script avr85xx.x
```

where `avr85xx.x` is the custom linker script (relative to the working directory). A custom script can be made by copying the default script from

```
~avr/avr/lib/ldscripts/avr85xx.x
```

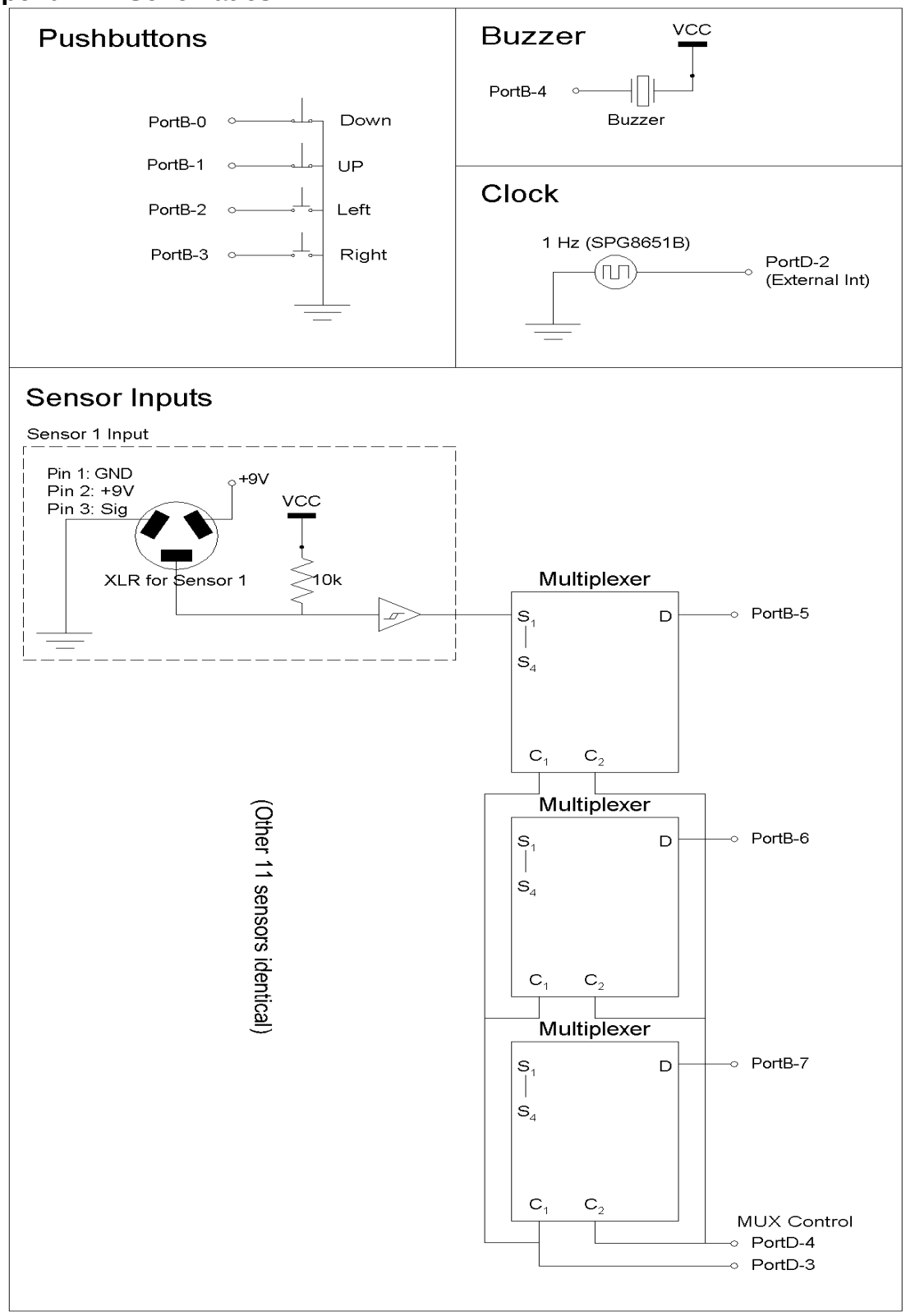
where `~avr` is the base directory for your AVR binutils, and modifying this script appropriately. The `MEMORY` section must be modified. The `LENGTH` parameter for the data memory section must be set to the total size of data memory (512b + 32K in this case),

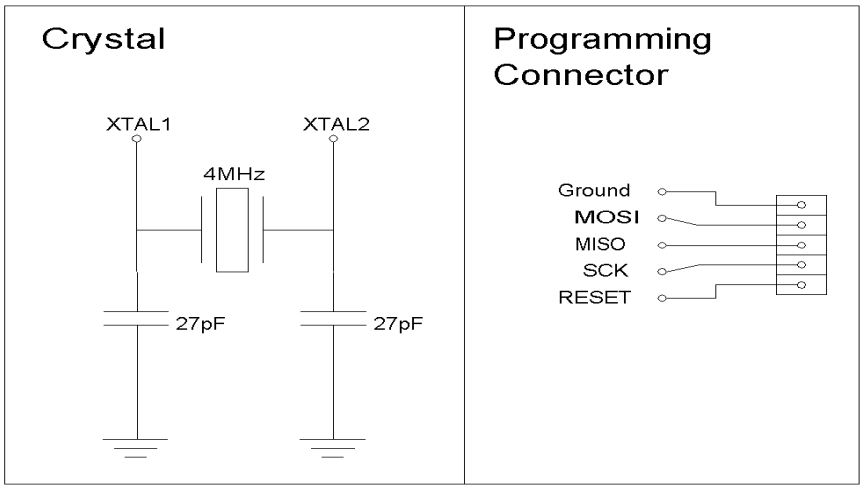
and the `ORIGIN` parameter must be increased by the size of internal memory (512b or 0x200). Setting the origin after internal memory ensures that all data memory will reside in external memory.

Note that the origin cannot be set to the start of internal memory. This would cause then the variable data to overlap the stack, which starts at the end of internal memory and grows downward. The controller will not function if the stack is moved out of internal memory, so the best thing to do is to have variable data begin right after the stack in external memory.

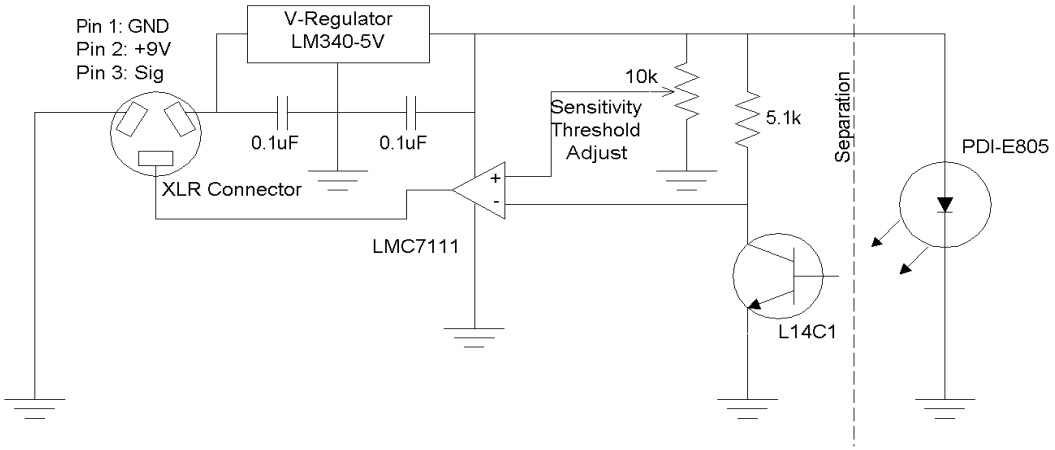
As of this writing, the website http://www.openavr.net is attempting to build a central repository of information for open-source AVR development tools. When I was setting up my Linux AVR development environment, it was difficult to find helpful information. This site may well develop into a useful resource.

Appendix E: Schematics



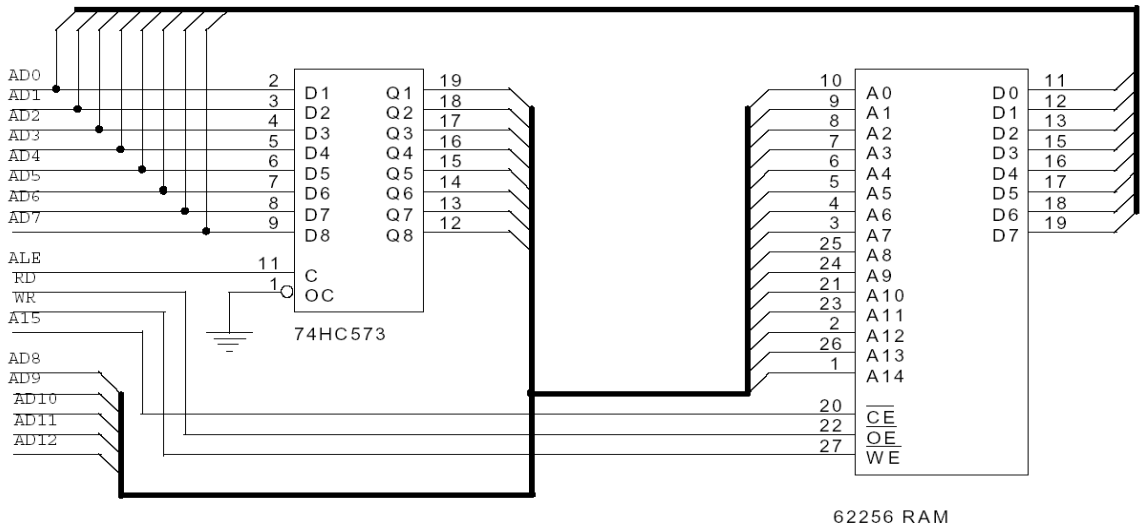


Sensor Circuitry

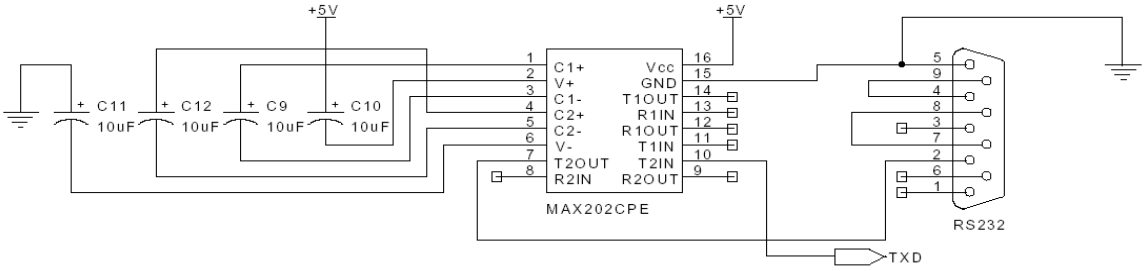


The following schematics were adapted from the data-sheet for the STK-200 development board.

Memory Connections



RS-232 Connection



The following schematic was adapted from the 90S8515 data-sheet from Atmel.

Controller Pinout