The BCD transient recorder revisited: firmware for the ISA-to-USB bridging card and software for the graphical user interface.

Mechanical Engineering Report 2007/06
P. A. Jacobs
Centre for Hypersonics
The University of Queensland.

March 25, 2007

Abstract

This report describes recent updates to the custom-built data-acquisition hardware operated by the Center for Hypersonics.

In 2006, an ISA-to-USB bridging card was developed as part of Luke Hillyard's final-year thesis. This card allows the hardware to be connected to any recent personal computers via a (USB or RS232) serial port and it provides a number of simple text-based commands for control of the hardware.

A graphical user interface program was also updated to help the experimenter manage the data acquisition functions. Sampled data is stored in text files that have been compressed with the gzip format. To simplify the later archiving or transport of the data, all files specific to a shot are stored in a single directory. This includes a text file for the run description, the signal configuration file and the individual sampled-data files, one for each signal that was recorded.

Contents

1 Introduction **3**

2 Description of the ISA-to-USB Bridging Card **4**

3 Using the GUI Management Program **6**
3.1 Maintenance of the Data Archives 10

A Firmware for the Bridging Card **13**
A.1 main.h 13
A.2 main.c 13
A.3 bits.h 15
A.4 isa-bus.h 15
A.5 isa-bus.c 15
A.6 uart-interrupt.h 17
A.7 uart-interrupt.c 17
A.8 databox.h 22
A.9 databox.c 23
A.10 Menu.txt 34

B Databox Management Program – Serial Version **36**
B.1 dbox_view_usb.py 36
B.2 dbox_services_usb.py 51
B.3 dbox_data_usb.py 58

1 Introduction

The Hypersonics Group at the University of Queensland has been operating a set of shock tunnels and expansion tubes for approximately twenty years. There are many thousands of data files collected from about 10000 shots, with most data (at UQ) being acquired by the locally-developed *BCD*[1] transient recorders (*a.k.a.* databoxes). The BCD databoxes were designed and constructed by the electronics lab staff (Barry Daniel, Barry Allsop and John Peters) in the late 1980s and early 1990s. Each box contains a power supply, modified-ISA backplane, a set of three time-base units, three trigger units and up to seven analogue-to-digital converter cards, each with three channels. There is also an external multiplexing system for combining up to four signals into one analogue-to-digital channel.

In 2006, Luke Hillyard designed and constructed a number of microcontroller-based bridging cards that plug into the ISA back-plane of the BCD databox and allowed control of the data acquisition process from any computer with a serial port. Both the classic RS232 port and the more recent USB port is supported by the card. This work is reported in Luke's final-year thesis [1].

The present report follows on from previous work [2] in which the register-level control of the databox was separated from the higher-level data-acquisition functions. Luke's embedded controller card now handles all of the register-level interaction with the transient recorder and it communicates with the management program via a set of simple text commands. The following sections provide the details of the the firmware for the bridging card and the updated GUI management program.

A companion report [3] is available online. It is a hypertext document containing the machine-readable form of the Tunnel-Data Server [4] software collection. This collection includes a client browser that can be used to display the collected data in a graphical format and a web-server component that can deliver data to client browsers across the internet.

[1] BCD are Barry Daniel's initials.

2 Description of the ISA-to-USB Bridging Card

The databox hardware is controlled by writing to and reading from a number of registers that are mapped to I/O ports of a host computer. Details of this register-level control can be found in report [2]. With the assistance of Barry Daniel, Luke Hillyard has designed and implemented an AVR microcontroller-based card that emulates enough of the PC's ISA bus to do the reading and writing of the appropriate registers in the databox. The firmware in Luke's thesis [1] has been rewritten and now appears in Appendix A.

One may now control the databox by plugging in any personal computer via a serial port. This serial port can be either a classic RS232 serial port or it can be the more modern USB type of serial port. If using the USB port, the FTDI chip used on the embedded card will register itself as a serial port on the personal computer[2]. The communication speed is permanently set at 230400 baud for the USB port and is selectable up to speeds of 115200 baud for the RS232 port.[3] Note that the other communications settings are 7-bit with 2 stop-bits and odd parity and that there is no hardware (CTS/RTS) or software (XON/XOFF) control of the communication. When sending large amounts of data back to the personal computer, check sums are used.

It is straight-forward to interact with the databox via a terminal program which could be `Hyperterm` on Windows or `minicom` on Linux. Commands to the databox are either one or two characters and the responses are in plain text. The following response from the "help" command describes the range of possible commands.

```

********************************* M E N U  *********************************

All uppercase and some lowercase commands require a following integer parameter i
e.g. R1 to reset
For convenience RR, AA, TT, & BB are equivalent to R1, A1, T1, & B1 respectively.
Card commands give a response for the selected card and channel unless otherwise indicated.

************************ A/D Card Control ************************

'Ri' : < i = 0 - 1 > : 1 = reset & hold ring pointers at location 0x0000
'Ni' : < i = 1 - F > : select card number
'Ci' : < i = 1 - 3 > : select channel number
'Di' : < i = 1 - 3 > : send header + 8k values for card N, channel i
   ** NOTE ** this command(Di) also sets Brief mode(B0) & selects Channel(Ci)
'Oi' : < i = 1 - 3 > : get the latest 15 bit value from channel i as octal
```

[2] On a Linux machine, a suitable device driver is most likely already present. On a Microsoft Windows computer, you will probably need to install a suitable FTDI device driver (http://www.ftdichip.com/FTDrivers.htm).

[3] The maximum speed has been programmed as the default RS232 baud rate in the current firmware.

```
'Vi' : < i = 1 - 3 > : get the latest conversion from channel i as BCD voltage
'f' : transmit the latest full scale data range (volts) (for channel NiCi)
'g' : transmit the latest data coupling (A or D) (for channel NiCi)

************************ A/D Card Status ************************

'xi' : < i = 1 - F > : check existence of card : 1 = appears to exist, 0 = not present
'ai' : < i = 1 - F > : get sampling status : 1 = still sampling, 0 = hold
'r' : get ring-buffer pointer to oldest data address : HHHH (hexadecimal)
't' : get card's timebase selection : 1-3 or 0 = timebase 1 at x4

********************* Trigger Unit Control *********************

'Ai' : < i = 0 - 1 > : 1 = arm all trigger units (0 = no effect)
'Ti' : < i = 0 - 1 > : 1 = causes a triggering of Databox (0 = no effect)

********************* Time Base Status *********************

'di' : < i = 1 - 3 > get thumbwheel selection : dd92 (decimal)
'bi' : < i = 1 - 3 > get buffer size selection : 2, 4, 8kb, A=Active
'pi' : < i = 1 - 3 > get sample period : 01-50microseconds
'mi' : < i = 1 - 3 > get multiplier : 1 = x100, 0 = x1
'ui' : < i = 1 - 3 > get timebase unit trigger # : (1-3)

********************* Trigger Unit Status **********************

'ci' : < i = 1 - 3 > get trigger unit coupling: D = 1 = dc, A = 0 = ac
'si' : < i = 1 - 3 > get slope setting: R = 1 = rising, F = 0 = falling
'ki' : < i = 1 - 3 > get trigger level percent : +/-99
'Li' : < i = 1 - 3 > get trigger level percent : octal
'zi' : < i = 0 - 1 > determine threshold-reporting behaviour: 1=enable, 0=disable

**************************** System Functions ****************************

'v' : get port and software version No.
'y' : check existence of box: 1 = appears to be present, 0 = not present
'i' : get programmed i/o base address: default=0x0320
'I' : < i = xxx > permanently change hex i/o base address. e.g I320
'Ui' : < i = 1 - 6 > permanently change RS232 baud rate
       1:2400 2:4800 3:9600 4:38400 5:57600 6:115200
'Bi' : < i = 0 - 1 > set output mode : 0 = brief, 1 = verbose
'h' : print this help menu
```

3 Using the GUI Management Program

User interaction with the BCD databoxes is handled via the Graphical User-Interface program `dbox_view_usb.py` (Appendix B). With the databox plugged into a convenient USB or RS232 serial port, open a terminal window and, within it, the GUI program

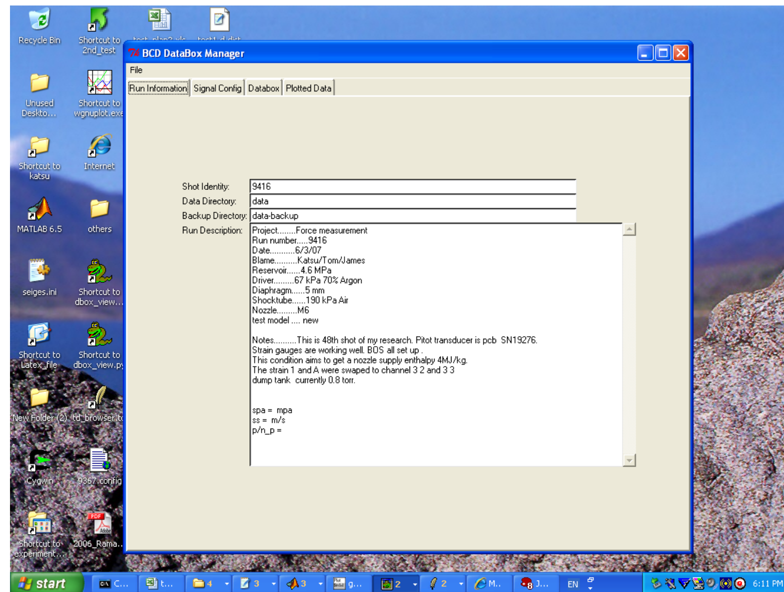
```
$ cd tds/databox/son_of_monc
$ ./dbox_view_usb.py
```

A desktop icon can also be set up for those people who like to double-click everything.

On startup, the program searches a number of serial ports, trying to find the databox. The search is stopped if a databox is found to be responding appropriately. If the search was unsuccessful, the program will still proceed to start up and be available to load previously recorded data. This might be useful in case where transducer sensitivities need to be altered some time after the experiment and data needs to be rescaled.

Once the program has completed setting up its GUI, the main window contains a notebook with 4 pages:

- **Data Management** indicates the shot identity (or number), where the data should be written and displays the run-description text in an edit widget. Choose a shot identity that matches the pattern decided for each particular facility. The T4 facility typically uses a 4-digit integer while the X2 and X3 facilities have been using the string `sNNN` where `NNN` is the integer shot number. The run description can be loaded or saved via entries in the `File` menu and, on saving the data, the content of the edit window is written to a text file in the shot directory. The following figure shows the management program as it was run on Katsuyoshi Tanimizu's MS-Windows computer in a recent scramjet test campaign.

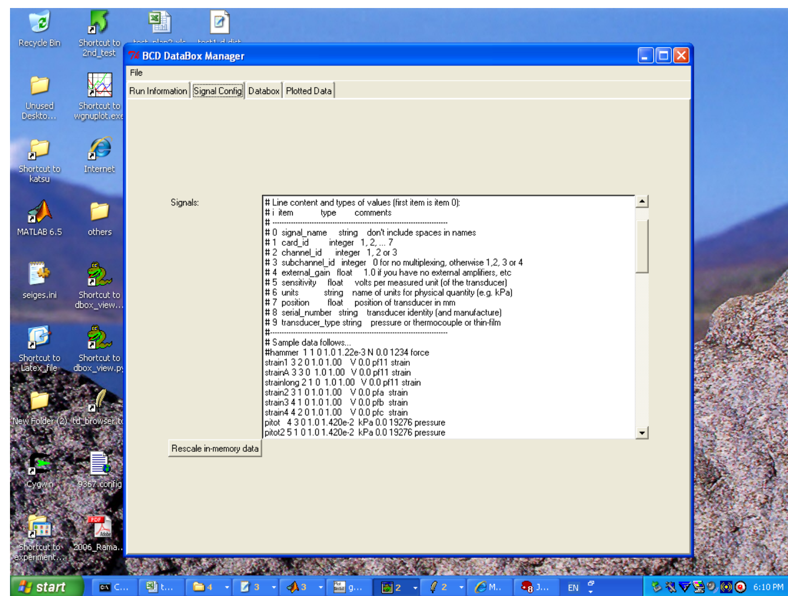


- **Signal Configuration** contains an edit widget with the text of the signal configuration file. Lines starting with "#" are comments and other lines, one per signal, must contain the following items, separated by white-space.

  0. signal name (a string of characters)
  1. card identity (integer 1..7)
  2. channel identity (integer 1..3)
  3. subchannel identity (1..4 for multiplexed signals, 0 otherwise). When specifying multiplexed signals, the software assumes that numbering starts at 1 and uses consecutive numbers. The subchannel with the highest initial value (over 20 samples) is assumed to be the first subchannel so offsets at the multiplexer input should be set appropriately.
  4. external gain (float). The external multiplexers may apply a non-unity gain before the transducer's signal reaches the databox.
  5. sensitivity (float) in volts per measured unit
  6. units (string) of the physical quantity. This value will appear in the y-axis label in the data browser [3] plot.
  7. position (float) in mm of the transducer. This value could be used for subsequent plotting of properties as a function of distance or for automating the estimation of shock speeds.

8. transducer identity (string). This could be the transducer's serial number or some combination of the serial number and the manufacturer name, for example.
9. type of signal (string). This may be one of "unknown", "pressure", "temperature", etc. Although the databox programs do not use this value, other programs that process the recorded data may attach significance to it.

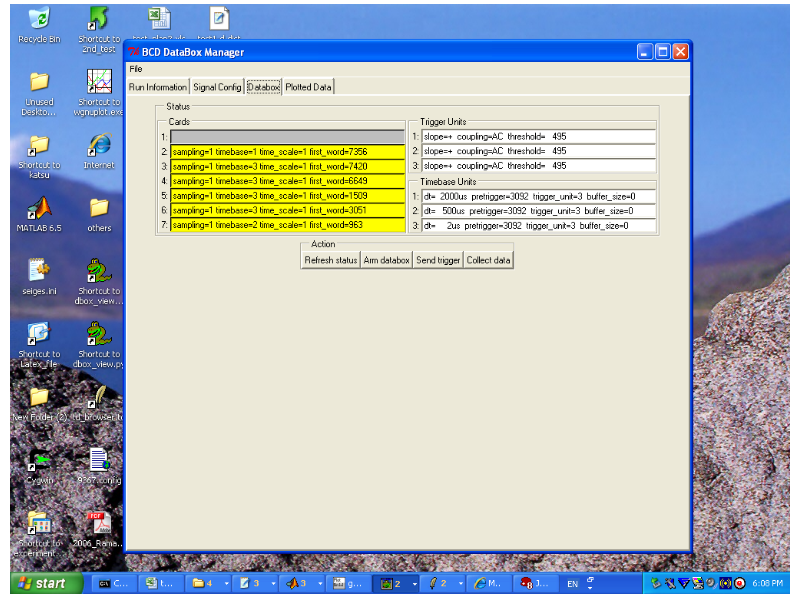
Note that it is *not* a good idea not to have embedded spaces in the items.



The text in the signal configuration edit widget can be loaded from or saved to an arbitrary file via entries in the `File` menu. On saving the data for the shot, the data is scaled according to the current sensitivity and external-gain values and written to the data files in the shot directory. Only scaled data is written, not the raw voltages. The user can load a previously recorded data set from disk using the entry in the `File` menu. The signal configuration of that saved data is also read and the raw voltages will be reconstructed from the scaled data. New scales can be applied by editing the sensitivity or external-gain values for a signal and pressing the `Rescale Data` button. This newly scaled data may be written to disk, over the old files or to a new shot identity, as directed by the user.

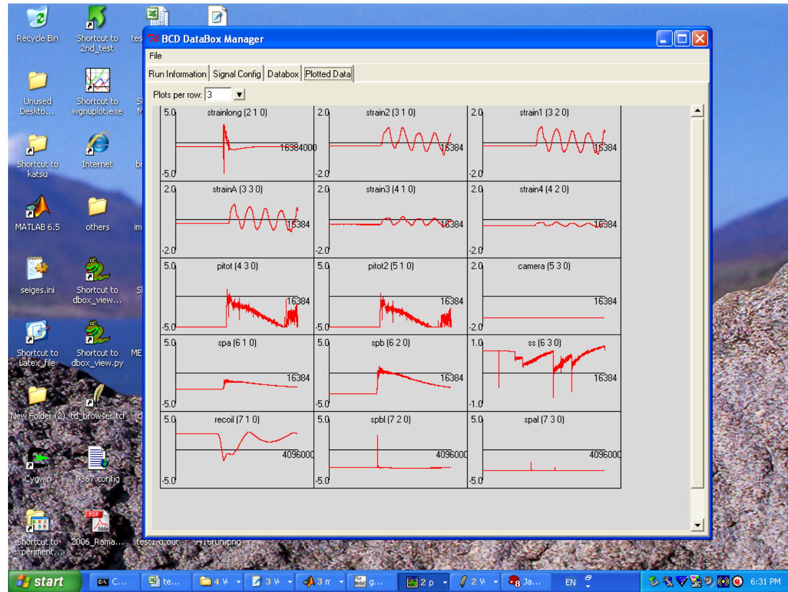
- **Databox Status and Control**: showing, in the top frame, the current state of the databox. The lower frame contains buttons for arming the databox and collecting the data. Be aware that the program is written in Python and the serial port is

used to pass a lot of text messages back and forth. This makes the software slower than the previous MONC program so the user needs to be patient when the full complement of channels is being used. Random activation of widgets during processing is likely to cause confusion, first to the program and then to the user.



A yellow background in a card-status widget indicates that the card is presently sampling data and is waiting for the trigger event.

- **Plots**: displays the raw voltage histories for the collected data. These represent the signals as they arrive at the input of the databox. Plots of the scaled data can be viewed with the `td_browser` program [3] or pressure versus distance plots can be made with the `px_view.py` program [2].



3.1 Maintenance of the Data Archives

The data archives as seen by a client computer are located on a web-server computer (`mech.uq.edu.au`) that is completely separate from the data-acquisition computers in the laboratory. It is the daily responsibility of individual experimenters to make sure that valid copies of their data files are put in the correct locations on the `triton` computer because the web-server archives are made from these files.

To use rsync to backup your newly collected files for T4 shot 9416 from a Linux machine to the master archive, use the rsync command

```
rsync -av -e ssh 9416 user@triton.pselab.uq.edu.au:/archive1/tunnels/T4/
```

with `user` replaced with a valid user name. The data is to be transferred via the secure-shell protocol and the triton system should challenge you for the password of the specified user name. After copying your new data files to the appropriate area on `triton`, make sure that the "group" identity of each shot directory and its contained data files is set to "tunnels". If you are logging into `triton`, use the `chgrp` command. From a MS-Windows machine, use a Secure-Shell client program to drag the shot directory into the appropriate tunnel directory on `triton`.

See section 4 of the previous report [2] for a description of the archive directory structure and the file format of the files for the data. The content of the files is text that has been compressed with standard compression utilities (gzip). It is easy to write your own analysis programs that either read or request the data from the web server and a couple

of example programs (in Python and MATLAB/Octave) have been provided in the code collection [3, 4].

References

[1] L. Hillyard. Embedded hardware manager for a high-speed data acquisition system. Final-year thesis, The University of Queensland, School of Information Technology and Electrical Engineering, October 2006.

[2] P. A. Jacobs. Databox manager: Device driver and user interface software. Department of Mechanical Engineering Report 2004/14, The University of Queensland, Brisbane, Australia., January 2005.

[3] P. A. Jacobs. TDS - shock tunnel data server and browser. Department of Mechanical Engineering Report 2002/08, The University of Queensland, Brisbane, Australia., January 2002. URL http://www.mech.uq.edu.au/cfcfd/.

[4] P. A. Jacobs. Tunnel-data server: Data format, archive maintenance and access protocol. Department of Mechanical Engineering Report 2003/06, The University of Queensland, Brisbane, Australia., May 2003.

A Firmware for the Bridging Card

The firmware for the bridging card is written in C using the WinAVR port of GCC (http://winavr.sourceforge.net) to Atmel's AVR Studio (http://www.atmel.com). Earlier versions were written by Luke Hillyard as part of his final-year thesis.

A.1 main.h

```
/* main.h */

/* Function prototypes */
void setBaudRate (void);
int main (void);
```

A.2 main.c

```
/* main.c
 * A project to provide an interface between the ISA bus and the Serial and
 * USB ports. Operating at the highest level this file maps characters
 * received from the serial port to functions that return data as required.
 *
 * 15/10/2006 Luke Hillyard
 * 16-Jan-2007 Peter J. -- code clean-up and comment
 * 17-Jan-2007 Peter J. -- some real restructuring in all of the files
 * 19-Jan-2007 Peter J. -- finalize and add checksum calculation
 * 24-Jan-2007 Peter J. -- rework assembly of data values from memory
 * 29-Jan-2007 Peter J. -- v3.3 Suppress reporting of threshold for T4 databox.
 * 05-Feb-2007 Peter J. -- v3.4 command to enable/disable threshold-reporting
 * 06-Feb-2007 Peter J. -- v3.5 move the menu text to program space so that we don't
 *                         consume all of the RAM with string data.
 */

#define VERSION "3.5"

#include <avr/io.h>
#include <avr/eeprom.h>
#include "isa-bus.h"
#include "databox.h"
#include "uart-interrupt.h"
#include "main.h"


unsigned char EEMEM baudRateSelection = 6; // 115200 baud
unsigned char verbose = 1; // Verbose = 1
unsigned char source = 1;  // RS232 Serial = 1
                           // USB Serial = 0

void setBaudRate () {
  // Reset the RS232 baud-rate from the selection received.
  eeprom_write_byte(&baudRateSelection, UART_Receivei());
  UART_Init(eeprom_read_byte(&baudRateSelection));
}

// ---------------------------------------------------------------------------

int main (void) {
  ISA_Init();
  UART_Init(eeprom_read_byte(&baudRateSelection));

  // The program is now driven by the data that comes down the serial line.
  for (;;) {
    switch ( UART_Receive() ) {
```

```
// All uppercase and some lowercase commands require aninteger parameter (i)
// e.g. R1 to reset //

// A/D Card Control
case 'R' : resetCardPointer(); break;  //<i=1-0> 0=reset & hold ring pointers at zero.
case 'N' : select_card(); break;  //<i=1-F> select card for other commands
case 'C' : select_channel(); break;  //<i=1-3> select channel for other commands

case 'D' : verbose = 0;   //force brief mode for all subsequent commands
       select_channel();
       getData(); break;  // send header+8k values for card N

case 'O' : octalOutput(); break;  //<i=1-3> transmit the latest 2 bytes as octal data
case 'V' : voltageOutput(); break;  //<i=1-3> transmit the latest 2 bytes as BCD data
case 'f' : fullScaleRange(); break; //transmit the latest data range
case 'g' : dataCoupling(); break;  //transmit the latest data coupling

// A/D Card Status
case 'x' : UART_Transmiti(card_is_present(UART_Receivei())); break; // sends '1' if card is presen
case 'a' : report_sampling_status(UART_Receivei()); break;  // 1=still sampling, 0=hold
case 't' : whichTimeBase(); break;    //switch: 1 to 3 or 0=timebase#1 x4
case 'r' : printBufferPointer(); break;  //get Ring buffer oldest data address,0xBBBB

// Trigger Unit Control
case 'A' : arm_box(); break;              //<i=1> 1=arm all trigger units
case 'T' : trigger_box(); break;    //<i=1> 1=causes a triggering of databox

// Time Base Status
case 'd' : preTriggerDelay(UART_Receivei()); break;  //get thumbwheels for time base 0xCD92
case 'b' : bufferSizeSelect(UART_Receivei()); break;  //get selected size (2,4,8kb,A=Active)
case 'p' : samplePeriod(UART_Receivei()); break;    //get sample period for timebase 01-50
case 'm' : timeBaseMultiplier(UART_Receivei());break;  //get multiplier 1=x100 0=x1

case 'u' : whichTriggerUnit(UART_Receivei()); break;

// Trigger Unit Status
case 'c' : report_coupling(UART_Receivei()); break;     // D=1=dc,A=0=ac
case 's' : report_slope(UART_Receivei()); break;        // 1=rising,0=falling
case 'L' : report_threshold(UART_Receivei()); break;    // in octal
case 'k' : report_threshold_in_BCD(UART_Receivei()); break; // decimal percentage +/- 99
case 'z' : enable_threshold_report(UART_Receivei()); break;    // 1=enable,0=disable

// System Functions
case 'y' : UART_Transmiti(box_is_present()); break; // sends '1' if databox is present, '0' other
case 'v' :
  if ( source && verbose ) {
    UART_Transmits("Data Box V" VERSION " Present on Serial Port");
  } else if ( source ) {
    UART_Transmits("S" VERSION);
  } else if ( verbose ) {
    UART_Transmits("Data Box V" VERSION " Present on USB Port");
  } else {
    UART_Transmits("U" VERSION);
  }
  if ( verbose ) {
    UART_Transmits("\r\n");
    UART_Transmits("Luke Hillyard: hardware and firmware, 2006.\r\n");
    UART_Transmits("Peter Jacobs: firmware, Jan 2007.\r\n");
  }
  break;
case 'i' : reportIOBase(); break; //get programmed i/o base address: default=0x0320
case 'I' : setIOBase(); break;    //set i/o base address. Needs 3 characters! e.g 320
                 // NOTE : BASE ADDRESS IS REMEMBERED AFTER POWER CYCLE
case 'U' : setBaudRate(); break;  // <i=1-6> set RS232 baud rate
                 // NOTE : BAUD RATE IS REMEMBERED AFTER POWER CYCLE
case 'B' : setVerbose(); break;   // <i=0-1> 0=brief 1=verbose
case 'h' : menu(); break;
```

```
      default:
        if ( verbose ) {
          UART_Transmits("What the fxck");
        }
        UART_Transmit('?');
        break;
    } // end switch()

    if (verbose) {
      UART_Transmits("\r\n");
    }
  } // end for()
  return 0;
}
```

A.3 bits.h

```
// bits.h

#ifndef __BITS_H
#define __BITS_H

// useful macros for setting indivdual bits
#define SETBIT(ADDRESS,BIT) (ADDRESS |= (1<<BIT))
#define CLEARBIT(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT))
#define CHECKBIT(ADDRESS,BIT) (ADDRESS & (1<<BIT))

#endif
```

A.4 isa-bus.h

```
// isa-bus.h

// Low Level Functions
void ISA_Init (void);
unsigned char IOread(unsigned int addr);
void IOwrite(unsigned int addr, unsigned char ISAData);
unsigned char MEMread(unsigned int addr);
```

A.5 isa-bus.c

```
// isa-bus.c
// Low level functions that fiddle the bits on the ISA bus of the databox.
//
// Luke Hillyard, October 2006
// Peter Jacobs, January 2007
//

#include <avr/io.h>
#include <avr/eeprom.h>
#include "bits.h"
#include "isa-bus.h"

// Addressing memory across the ISA bus.
                            // X1 (0 = on; 1 = off)
// 0xD8000 Databox -->> 0b 1101 1000 0000 0000 0000
// Interface Card  -->> 0b xxxx xx00 0000 0000 0000
// With the way that the hardware is built,
// there is now no way to select anything other than
// 0b 1101 10xx xxxx xxxx
// 0x D    8    0    0

#define DATABUSDIR DDRF
#define INPUT 0x00
#define OUTPUT 0xFF
#define IORC PIN1
```

```
#define IOWC PIN0
#define SMRDC PIN3

void ISA_Init(void) {
  DDRA = 0xFF;        //Address Bus Low-order bits
  // Use line below to mask out SA14 and SA15 (PINC6 and PINC7)
  DDRC = 0b00111111; //Address Bus High-order bits
  //
  // Control Lines - HIGH is the ISA default
  SETBIT(PORTG, SMRDC);
  SETBIT(DDRG, SMRDC);
  SETBIT(PORTG, IORC);
  SETBIT(DDRG, IORC);
  SETBIT(PORTG, IOWC);
  SETBIT(DDRG, IOWC);
  return;
}

unsigned char IOread(unsigned int addr) {
  // Return a byte from the ISA bus, IO area.
  //
  unsigned char ISAData;
  //
  // Set up the address pattern for the subset of the ISA bus
  // that we are using in the databox.
  PORTA = addr & 0xFF;
  PORTC = (addr>>8) & 0xFF;
  //
  DATABUSDIR = INPUT;
  CLEARBIT(PORTG,IORC);
  // Insert a 0.325us wait.
  // This should be long enough for Barry's electronics to have settled.
  asm volatile("NOP"::); // at 18.432MHz, each NOP represents 54.2ns
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  ISAData = PINF;
  SETBIT(PORTG,IORC);
  return ISAData;
}

void IOwrite(unsigned int addr, unsigned char ISAData) {
  // Write a byte (ISAData) onto the ISA bus as IO data.
  //
  // Set up the address pattern for the subset of the ISA bus
  // that we are using in the databox.
  PORTA = addr & 0xFF;
  PORTC = (addr>>8) & 0xFF;
  //
  PORTF = ISAData;
  DATABUSDIR = OUTPUT;
  CLEARBIT(PORTG,IOWC);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  SETBIT(PORTG,IOWC);
  // PORTG = 0xFF; // Faster and the default is high anyway.
  DATABUSDIR = INPUT;
  return;
}

unsigned char MEMread(unsigned int addr) {
  // Returns a byte from the ISA memory area.
```

```
  //
  unsigned char ISAData;
  //
  // Set up the address pattern for the subset of the ISA bus
  // that we are using in the databox.
  PORTA = addr & 0xFF;
  PORTC = (addr>>8) & 0xFF;
  //
  DATABUSDIR = INPUT;
  CLEARBIT(PORTG,SMRDC);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  asm volatile("NOP"::);
  ISAData = PINF;
  SETBIT(PORTG,SMRDC);
  return ISAData;
}
```

A.6 uart-interrupt.h

```
/* uart-interrupt.h */

#ifndef __UART_INTERRPUT_H
#define __UART_INTERRUPT_H

void UART_Init (unsigned char rs232_baud_rate_selection);
void UART_Transmit0 (unsigned char data);
void UART_Transmit1 (unsigned char data);
unsigned char UART_Receive0 (void);
unsigned char UART_Receive1 (void);

void UART_Transmit (unsigned char data);
unsigned char UART_Receive (void);

void UART_TransmitBCD (unsigned int data);
void UART_Transmiti (unsigned char data);
void UART_Transmitlong (unsigned int data);
void UART_Transmits (unsigned char* data);
void UART_Transmitb (unsigned char data);

int UART_Receivei (void);

#endif
```

A.7 uart-interrupt.c

```
/* UART-INTERRUPT.C
 * An implmentation of a interrupt driven UART for the Databox Interface
 * Project. Uses circular buffers for RX and TX and automatically merges data
 * from both UARTS.
 *
 * 15/10/2006 Luke Hillyard
 * 16-Jan-2007 Peter J.  clean-up and comment
 */

#include <avr/interrupt.h>
#include <avr/io.h>
#include <avr/eeprom.h>
#include "bits.h"
#include "databox.h"
#include "uart-interrupt.h"

//Constants for Baud Rate Calculation
extern unsigned char source;
```

```
//Size of RX and TX buffers in bytes (chars)
#define Uart0TXBuffSizeMax 64
#define Uart0RXBuffSizeMax 64
#define Uart1TXBuffSizeMax 64
#define Uart1RXBuffSizeMax 64

//RX and TX buffers;
volatile char Uart0TXBuff[Uart0TXBuffSizeMax];
volatile char Uart0RXBuff[Uart0RXBuffSizeMax];
volatile char Uart1TXBuff[Uart1TXBuffSizeMax];
volatile char Uart1RXBuff[Uart1RXBuffSizeMax];

//Variables to track the size of the buffers
volatile unsigned char Uart0TXBuffSize = 0;
volatile unsigned char Uart0RXBuffSize = 0;
volatile unsigned char Uart1TXBuffSize = 0;
volatile unsigned char Uart1RXBuffSize = 0;

//Pointer to Unread / Unsend data
volatile unsigned char Uart0TXBuffLoc = 0;
volatile unsigned char Uart0RXBuffLoc = 0;
volatile unsigned char Uart1TXBuffLoc = 0;
volatile unsigned char Uart1RXBuffLoc = 0;

//Pointer to next free location in buffer
volatile unsigned char Uart0TXBuffInsertLoc = 0;
volatile unsigned char Uart0RXBuffInsertLoc = 0;
volatile unsigned char Uart1TXBuffInsertLoc = 0;
volatile unsigned char Uart1RXBuffInsertLoc = 0;


// -------------- Interrupt Handlers ------------------

SIGNAL (SIG_UART0_RECV) {
  // A character has been received from the USB port; put it in the receive buffer.
  cli();
  if (Uart0RXBuffSize < Uart0RXBuffSizeMax) {
    // Space is available
    Uart0RXBuffSize++;
    Uart0RXBuff[Uart0RXBuffInsertLoc] = UDR0;
    Uart0RXBuffInsertLoc++;
    if (Uart0RXBuffInsertLoc == Uart0RXBuffSizeMax) {
      Uart0RXBuffInsertLoc = 0;
    }
  } else {
    /* do nothing */ ; // Lose the character
  }
  sei();
}

SIGNAL (SIG_UART0_DATA) {
  // The transmit side of UART0 is waiting for more data.
  // Send a character if there is one in the buffer,
  // else tell UART0 to stop interrupting.
  cli();
  // Presumably, we have data to send.
  UDR0 = Uart0TXBuff[Uart0TXBuffLoc];
  Uart0TXBuffSize--;
  Uart0TXBuffLoc++;
  if (Uart0TXBuffLoc == Uart0TXBuffSizeMax) {
    Uart0TXBuffLoc = 0;
  }
  if (Uart0TXBuffSize == 0) {
    // There is no more data to send; tell UART0 to stop interrupting.
    CLEARBIT(UCSR0B,UDRIE);
  }
  sei();
```

```
}

SIGNAL (SIG_UART1_RECV) {
  // A character has been received at UART1; put it in the receive buffer.
  cli();
  if (Uart1RXBuffSize < Uart1RXBuffSizeMax) {
    // There is space available.
    Uart1RXBuff[Uart1RXBuffInsertLoc] = UDR1;
    Uart1RXBuffSize++;
    Uart1RXBuffInsertLoc++;
    if (Uart1RXBuffInsertLoc == Uart1RXBuffSizeMax) {
      Uart1RXBuffInsertLoc = 0;
    }
  } else {
    /* do nothing */ ; // Lose the character.
  }
  sei();
}

SIGNAL (SIG_UART1_DATA) {
  // The transmit side of UART1 is waiting for more data.
  // Send a character if there is one in the buffer,
  // else tell UART1 to stop interrupting.
  cli();
  // Presumably, we have data to send.
  UDR1 = Uart1TXBuff[Uart1TXBuffLoc];
  Uart1TXBuffSize--;
  Uart1TXBuffLoc++;
  if (Uart1TXBuffLoc == Uart1TXBuffSizeMax) {
    Uart1TXBuffLoc = 0;
  }
  if (Uart1TXBuffSize == 0) {
    // No data to send; tell UART1 to stop interrupting.
    CLEARBIT(UCSR1B,UDRIE);
  }
  sei();
}

// ------------- functions that deal directly with hardware -------------

void UART_Init (unsigned char rs232_baud_rate_selection) {
  // The USART Initialisation routine
  //
  // Both UARTs:
  //    Asynchronous Normal Mode
  //    7 Data Bits, Odd Parity, 2 Stop Bits
  //    No Flow Control
  // Set the baud rates for each of the UARTs:
  #define FOSC 18432000L
  unsigned int UBRR;
  unsigned long rs232_baud_rate;

  cli(); // we don't want to be interrupted while fiddling UART config bits.
  // UART0 (connected to USB) at 230.4k.
  #define BAUD 230400
  UBRR0H = (unsigned char)((FOSC/16/BAUD-1)>>8);
  UBRR0L = (unsigned char)(FOSC/16/BAUD-1);
  /* 7 Bit mode */
  CLEARBIT(UCSR0B, UCSZ02);
  SETBIT(UCSR0C, UCSZ01);
  CLEARBIT(UCSR0C, UCSZ00);
  /* 2 Stop bit */
  SETBIT(UCSR0C, USBS0);
  /* Odd Parity */
  SETBIT(UCSR0C, UPM01);
  SETBIT(UCSR0C, UPM00);
  /* Enable receiver and transmitter */
  SETBIT(UCSR0B,RXEN);  //Enable Recieve
```

```
  SETBIT(UCSR0B,TXEN);   //Enable Transmit
  SETBIT(UCSR0B,RXCIE); //Interupt when data recieved

  // UART1 (connected via RS232) baud-rate encoded in UBRR.
  switch (rs232_baud_rate_selection) {
    case 1: rs232_baud_rate = 2400; break;
    case 2: rs232_baud_rate = 9600; break;
    case 3: rs232_baud_rate = 19200; break;
    case 4: rs232_baud_rate = 38400; break;
    case 5: rs232_baud_rate = 57600; break;
    case 6: rs232_baud_rate = 115200L; break;
    default: rs232_baud_rate = 57600;
  }
  UBRR = (FOSC/16/rs232_baud_rate - 1);

  UBRR1H = (unsigned char)(UBRR>>8);
  UBRR1L = (unsigned char)UBRR;
  /* 7 Bit mode */
  CLEARBIT(UCSR1B, UCSZ12);
  SETBIT(UCSR1C, UCSZ11);
  CLEARBIT(UCSR1C, UCSZ10);
  /* 2 Stop bit */
  SETBIT(UCSR1C, USBS1);
  /* Odd Parity */
  SETBIT(UCSR1C, UPM11);
  SETBIT(UCSR1C, UPM10);
  /* Enable receiver and transmitter */
  SETBIT(UCSR1B,RXEN);   //Enable Recieve
  SETBIT(UCSR1B,TXEN);   //Enable Transmit
  SETBIT(UCSR1B,RXCIE); //Interupt when data recieved

  sei(); // let the interrupt routines take over.
} // end UART_Init()


void UART_Transmit0 (unsigned char data) {
  // Send a character to the USB port circular buffer.
  while (Uart0TXBuffSize >= Uart0TXBuffSizeMax) {
    // The TX buffer is full; let the hardware deal with it.
    ; // Wait
  }
  cli();
  Uart0TXBuff[Uart0TXBuffInsertLoc] = data;
  Uart0TXBuffSize++;
  Uart0TXBuffInsertLoc++;
  if (Uart0TXBuffInsertLoc == Uart0TXBuffSizeMax) {
    Uart0TXBuffInsertLoc = 0;
  }
  // Since there is now data to send, tell UART1 to signal when it is ready.
  SETBIT(UCSR0B,UDRIE);
  sei();
}

void UART_Transmit1 (unsigned char data) {
  // Transmit a character to the RS232 serial port.
  while (Uart1TXBuffSize >= Uart1TXBuffSizeMax) {
    // The TX buffer is full; let the hardware deal with it.
    ; // Wait
  }
  cli();
  Uart1TXBuff[Uart1TXBuffInsertLoc] = data;
  Uart1TXBuffSize++;
  Uart1TXBuffInsertLoc++;
  if (Uart1TXBuffInsertLoc == Uart1TXBuffSizeMax) {
    Uart1TXBuffInsertLoc = 0;
  }
  // Since there is now data to send, tell UART1 to signal when it is ready.
  SETBIT(UCSR1B,UDRIE);
```

```
  sei();
}

unsigned char UART_Receive0 (void) {
  // Receive a character from the USB port.
  unsigned char data;
  while (Uart0RXBuffSize == 0) {
    // The receive buffer is empty; wait for the hardware to put something in it.
    /* do nothing */ ;
  }
  cli();
  data = Uart0RXBuff[Uart0RXBuffLoc];
  Uart0RXBuffSize--;
  Uart0RXBuffLoc++;
  if (Uart0RXBuffLoc == Uart0RXBuffSizeMax) {
    Uart0RXBuffLoc = 0;
  }
  sei();
  return data;
}

unsigned char UART_Receive1 (void) {
  // Receive a character from the RS232 port.
  unsigned char data;
  while (!Uart1RXBuffSize) {
    // The receive buffer is empty; wait for the hardware to put something in it.
    /* do nothing */ ;
  }
  cli();
  data = Uart1RXBuff[Uart1RXBuffLoc];
  Uart1RXBuffSize--;
  Uart1RXBuffLoc++;
  if (Uart1RXBuffLoc == Uart1RXBuffSizeMax) {
    Uart1RXBuffLoc = 0;
  }
  sei();
  return data;
}

// ------------ functions to unify the UART streams --------------

void UART_Transmit (unsigned char data) {
  // Send a character to the appointed serial port.
  if (source) {
    UART_Transmit1(data);
  } else {
    UART_Transmit0(data);
  }
}

unsigned char UART_Receive (void) {
  // Hang around until we get one character from either UART.
  for (;;) {
    if (Uart0RXBuffSize) {
      source = 0;
      return UART_Receive0();
    }
    if (Uart1RXBuffSize) {
      source = 1;
      return UART_Receive1();
    }
  } // end for
}

// ------ higher-level functions to deal with various data types ------

void UART_TransmitBCD (unsigned int V) {
  // Send a 16-bit fixed-point value as a decimal digits,
```

```
  // starting with the most significant.
  unsigned int i;
  unsigned char C;
  for (i = 10000; i >= 1; i = i / 10) {
    C = V / i;
    V = V - C * i;
    UART_Transmiti(C);
    if (i == 10000) {
      UART_Transmit('.');
    }
  }
}

void UART_Transmiti (unsigned char data) {
  // Send a binary value encoded as a single hexadecimal digit.
  unsigned char temp;
  temp = (unsigned char)(data & 0xF);
  if (temp < 10) {
    temp = temp + '0';
  } else {
    temp = temp + 'A' - 10;
  }
  UART_Transmit(temp);
}

void UART_Transmitlong (unsigned int data) {
  // Send a 16-bit integer as 4 hexadecimal digits, one for each nibble,
  // starting with the most significant.
  char temp;
  char i;
  for (i = 3; i >= 0; i--) {
    temp = (data >> (4*i)) & 0xF;
    if (temp < 10) {
      temp = temp + '0';
    } else {
      temp = temp + 'A' - 10;
    }
    UART_Transmit(temp);
  }
}

void UART_Transmits (unsigned char* data) {
  // Send a null-terminated string (but not the null character itself).
  while (*data != '\0') {
    UART_Transmit(*data++);
  }
}

void UART_Transmitb (unsigned char data) {
  // Send a byte, one bit at a time, encoded as ASCII '1' or '0'.
  char i;
  for (i = 7; i >= 0; i--) {
    UART_Transmit(((data >> i) & 0b1) + '0');
  }
}

int UART_Receivei (void) {
  // Retrieve one hexadecimal digit from the UART and return its binary value.
  unsigned char temp = UART_Receive();
  if (temp >= 'A') {
    return (int)((temp - 'A' + 10) & 0xF);
  } else {
    return (int)((temp - '0') & 0xF);
  }
}
```

A.8 databox.h

```
/* databox.h */

#ifndef _DATABOX_H
#define _DATABOX_H

// A/D Card Control
void writeCardControlRegister(void);
void resetCardPointer(void);
void select_channel(void);
void select_card(void);
void send_encoded_word(unsigned int);
unsigned int rotate_right(unsigned int c);
void getData(void);
void octalOutput(void);
void voltageOutput(void);
void fullScaleRange(void);
void dataCoupling(void);

// A/D Card Status
unsigned char readCardStatusRegisterHigh(unsigned char n);
unsigned char readCardStatusRegisterLow(unsigned char n);
unsigned char card_is_present(unsigned char n);
unsigned char card_is_sampling(unsigned char n);
unsigned char any_cards_sampling(void);
void report_sampling_status(unsigned char n);
unsigned char whichTimeBase(void);
int getBufferPointer(void);
void printBufferPointer(void);

// Trigger Control
void writeTriggerUnitControlRegister(unsigned char ISAData);
void arm_box(void);
void trigger_box(void);

// Time Base Status
unsigned char readTimeStatusRegisterHigh(unsigned char n);
unsigned char readTimeStatusRegisterLow(unsigned char n);
unsigned char box_is_present(void);
void trigger_number(unsigned char n);
void preTriggerDelay(unsigned char n);
void bufferSizeSelect(unsigned char n);
unsigned char whichTriggerUnit(unsigned char n);
void samplePeriod(unsigned char n);
void timeBaseMultiplier(unsigned char n);

// Trigger Unit Status
unsigned char readTriggerStatusRegisterHigh(void);
unsigned char readTriggerStatusRegisterLow(void);
void report_coupling(unsigned char n);
void report_slope(unsigned char n);
void reportSlopeAndCouple(unsigned char n);
void enable_threshold_report(unsigned char n);
void report_threshold(unsigned char n);
void report_threshold_in_BCD(unsigned char n);
unsigned char thresholdStatus(void);

// System Functions
void reportIOBase(void);
void setIOBase(void);
void setVerbose(void);
void menu(void);

#endif
```

A.9 databox.c

```
/* databox.c
 *
```

```
 * An set of instructions used specifically for the Databox Interface Card.
 * This file implements the high level user functionality required by the
 * end user.
 *
 * Luke Hillyard, 15/10/2006
 * Peter Jacobs,  12-Jan-2007
 */

#include <avr/io.h>
#include <avr/eeprom.h>
#include <avr/pgmspace.h>
#include "bits.h"
#include "isa-bus.h"
#include "databox.h"
#include "uart-interrupt.h"

// Addressing Databox Registers
int EEMEM IOBaseAddress   = 0x320;    // 110
int EEMEM TimeBaseAddress = 0x310;

// Most of the operations on the A/D cards work on the
// currently selected card.
unsigned char CardNum = 1;    // 1 to 15 inclusive
unsigned char ChannelNum = 1; // 1 to 3  inclusive
unsigned char CardReset = 0;
extern unsigned char verbose;

unsigned char suppress_k_report = 1;

// ------------------ A/D Card Control -------------------------

void writeCardControlRegister(void) {
  // Writes byte to the card control register and then sends
  // corresponding text to the serial port.
  unsigned char ISAData = (CardReset << 7) | (ChannelNum << 4) | CardNum;
  IOwrite(eeprom_read_word(&IOBaseAddress) - 1, ISAData);
}

void resetCardPointer(void) {
  // Resets the circular buffer data pointer for the selected card.
  if (UART_Receivei()) {
    CardReset = 1;
  } else {
    CardReset = 0;
  }
  writeCardControlRegister();
  if (verbose) {
    UART_Transmits("Card Number : ");
    UART_Transmiti(CardNum);
    UART_Transmits(", Channel Number : ");
    UART_Transmiti(ChannelNum);
    UART_Transmits(" is selected & buffer pointer is ");
    if (CardReset) {
      UART_Transmits("held at 0 ");
    } else {
      UART_Transmits("free to increment ");
    }
  }
}

void select_channel(void) {
  // Sets the selected channel.
  ChannelNum = UART_Receivei();
  writeCardControlRegister();
  if (verbose) {
    UART_Transmits("Card Number : ");
    UART_Transmiti(CardNum);
    UART_Transmits(", Channel Number : ");
```

```
    UART_Transmiti(ChannelNum);
    UART_Transmits(" is selected & buffer pointer is ");
    if (CardReset) {
      UART_Transmits("held at 0 ");
    } else {
      UART_Transmits("free to increment ");
    }
  }
}

void select_card(void) {
  // Sets the selected card.
  CardNum = UART_Receivei();
  writeCardControlRegister();
  if (verbose) {
    UART_Transmits("Card Number : ");
    UART_Transmiti(CardNum);
    UART_Transmits(", Channel Number : ");
    UART_Transmiti(ChannelNum);
    UART_Transmits(" is selected & buffer pointer is ");
    if (CardReset) {
      UART_Transmits("held at 0 ");
    } else {
      UART_Transmits("free to increment ");
    }
  }
}

void send_encoded_word(unsigned int data) {
  // The 16-bit binary data really only contains 12 bits of useful data.
  // Break this into two 6-bit values and add 0x30 to each.
  // Each character will then be contained within 7-bits and
  // be in the range of printable characters.
  UART_Transmit((data & 0b111111) + 0x30);
  UART_Transmit(((data >> 6) & 0b111111) + 0x30);
}

unsigned int rotate_right( unsigned int c ) {
  // rotate-right with carry for 16-bit quantity
  if ( c & 1 )
    return (c>>1) | 0x8000;
  else
    return c>>1;
}

void getData(void) {
  // Sends the data for the selected channel on the selected card.
  // The full packet consists of a 20-byte header, a 16-kbyte payload
  // and an 8-byte trailer.
  //
  //
  if ( card_is_sampling(CardNum) ) {
    UART_Transmits("FAILED");
    return;
  }
  //
  writeCardControlRegister();
  if (verbose) {
    UART_Transmits("Card Number : ");
    UART_Transmiti(CardNum);
    UART_Transmits(", Channel Number : ");
    UART_Transmiti(ChannelNum);
    UART_Transmits(" is selected & buffer pointer is ");
    if (CardReset) {
      UART_Transmits("held at 0 ");
    } else {
      UART_Transmits("free to increment ");
    }
```

```
}
//
// Header:
// 1 Card Number = (1 -> 5)
UART_Transmiti(CardNum);
// 2 Channel Number = (1 -> 3)
UART_Transmiti(ChannelNum);
// 3 Timebase = (0 -> 3)
unsigned char n_timebase = whichTimeBase();
// 4 Period (us) High Byte (BCD)
// 5 Period (us) Low Byte (BCD)
samplePeriod(n_timebase);
// 6 Multiplier = (0 -> 1)
timeBaseMultiplier(n_timebase);
// 7 Pretrigger Setting High Byte (BCD) = (0 -> 9)
// 8 Pretrigger Setting Byte (BCD) = (0 -> 7)
// 9 Pretrigger Setting Byte (BCD) = 9
// 10 Pretrigger Setting Low Byte (BCD) = 2
preTriggerDelay(n_timebase);
// 11 Buffer Switch = (0 ,2 ,4, 8)
bufferSizeSelect(n_timebase);
// 12 Trigger = (1 -> 3)
unsigned char n_trigger = whichTriggerUnit(n_timebase);
// 13 Raw Slope and Coupling ( S C ) (0 -> 3)
reportSlopeAndCouple(n_trigger);
// 14 Trigger Level % ( + / - )
// 15 Trigger Level % (tens, BCD)
// 16 Trigger Level % (ones, BCD)
report_threshold_in_BCD(n_trigger);
// 17 DATA COUPLING A/D
dataCoupling();
// 18 FULL SCALE DATA RANGE 0 - 5
// 19 FULL SCALE DATA RANGE .
// 20 FULL SCALE DATA RANGE 5 - 0
fullScaleRange();
//
// Payload of 16-kbytes
// from oldest data
//     21 B B b b b b + 0x30
//     22 B B B B B B + 0x30
//   ...
// to newest data
// 16403 B B b b b b + 0x30
// 16404 B B B B B B + 0x30
int buf = getBufferPointer();
unsigned int data;
unsigned char low_byte, high_byte;
unsigned int i;
unsigned int chksum = 0;
// Start with the oldest data.
for ( i = buf; i < 8192; i++ ) {
  low_byte = MEMread(2 * i);
  high_byte = MEMread(2 * i + 1);
  // Assemble a 12-bit integer value from the two bytes.
  data = ((unsigned int)low_byte >> 4) | ((unsigned int)high_byte << 4);
  send_encoded_word(data);
  // accumulate 16-bit checksum
  chksum = rotate_right( chksum );
  chksum += data;
  chksum &= 0xffff; // probably redundant for AVR
}
// Wrap around to the newest data.
for ( i = 0; i < buf; i++ ) {
  low_byte = MEMread(2 * i);
  high_byte = MEMread(2 * i + 1);
  data = ((unsigned int)low_byte >> 4) | ((unsigned int)high_byte << 4);
  send_encoded_word(data);
  // accumulate 16-bit checksum
```

```
    chksum = rotate_right( chksum );
    chksum += data;
    chksum &= 0xffff; // probably redundant for AVR
  }
  //
  // Trailer
  UART_Transmitlong(chksum); // 16405-16408
  UART_Transmits("zzzz");    // 16409-16412 reserved
} // end getData()

unsigned int newestByte(void) {
  // Returns the most-recent word from the circular buffer.
  int buf = getBufferPointer();
  int latest;
  unsigned char high_byte, low_byte;
  unsigned int Data;
  switch (buf) {
    case 0 : latest = 16383; break;
    default : latest = (2 * buf) - 1;
  }
  high_byte = MEMread(latest);
  Data = (unsigned int)high_byte << 8;
  low_byte = MEMread(latest - 1);
  Data = (unsigned int)low_byte | Data;
  return Data;
}

void octalOutput(void) {
  // Sends an octal representation of the most-recent voltage
  // of the currently selected card and channel.
  //
  // If a card is not present (or functioning), we will most likely
  // get all high bits and an octal value of 77777.
  select_channel();
  unsigned int latest = newestByte();
  unsigned char i;
  for ( i = 5; i > 0; i--) {
    UART_Transmiti(((latest >> (3 * i - 2)) & 0b111));
  }
}

void voltageOutput(void) {
  // Sends a decimal representation of the most-recent voltage
  // of the currently selected card and channel.
  select_channel();
  unsigned int latest = newestByte();
  unsigned int mantissa = latest >> 4;
  if (mantissa > 2047) {
    UART_Transmit('+');
    mantissa = mantissa - 2048;
  } else {
    UART_Transmit('-');
    mantissa = 2048 - mantissa;
  }
  latest = (latest >> 1) & 0x7;
  mantissa = (mantissa * 31 + mantissa / 4) >> 5;
  switch (latest) {
    case 0 : mantissa = 60000; break;
    case 1 : mantissa = mantissa * 25; break;
    case 2 : mantissa = mantissa * 10; break;
    case 3 : mantissa = mantissa * 5; break;
    case 4 : mantissa = (mantissa * 5) / 2; break;
    case 5 : mantissa = mantissa * 1; break;
    case 6 :
    case 7 : mantissa = mantissa / 2; break;
  }
  UART_TransmitBCD(mantissa);
}
```

```
void fullScaleRange(void) {
  // Sends the full-scale voltage range as a fixed-point decimal number.
  unsigned int latest = (newestByte() >> 1) & 0b111 ;
  int c1[8] = {0, 5, 2, 1, 0, 0, 0, 0};
  int c2[8] = {0, 0, 0, 0, 5, 2, 1, 0};
  UART_Transmiti(c1[latest]);
  UART_Transmit('.');
  UART_Transmiti(c2[latest]);
}

void dataCoupling(void) {
  // Sends a text representation of the voltage coupling (AC/DC).
  unsigned int latest = (newestByte () & 0b1);
  if (latest) {
    UART_Transmit('D');
  } else {
    UART_Transmit('A');
  }
  if (verbose) {
    UART_Transmits("C Data Coupling");
  }
}


// ——————————— A/D Card Status ————————————

unsigned char readCardStatusRegisterHigh(unsigned char n) {
  return IOread(eeprom_read_word(&IOBaseAddress) + 2 * n + 1);
}

unsigned char readCardStatusRegisterLow(unsigned char n) {
  return IOread(eeprom_read_word(&IOBaseAddress) + 2 * n);
}

unsigned char card_is_present(unsigned char n) {
  // Returns 1 if card n appears to be present, 0 otherwise.
  // If a card is missing, the ISA bus terminators will pull
  // the data lines high.
  unsigned char all_ones = (readCardStatusRegisterHigh(n) == 0xFF) &&
                           (readCardStatusRegisterLow(n) == 0xFF);
  return !all_ones;
}

unsigned char card_is_sampling(unsigned char n) {
  // Returns 1 is still sampling, 0 otherwise.
  unsigned char ISAData = readCardStatusRegisterHigh(n);
  return ISAData >> 7;
}

unsigned char any_cards_sampling(void) {
  // Returns 1 if any existing cards are sampling, 0 otherwise.
  unsigned char any_sampling = 0;
  unsigned char i;
  for ( i = 1; i <= 7; ++i ) {
    if ( card_is_present(i) ) {
      if ( card_is_sampling(i) ) any_sampling = 1;
    }
  }
  return any_sampling;
}

void report_sampling_status(unsigned char n) {
  // Sends text indicating if the card is still sampling.
  if (verbose) {
    UART_Transmits("Card Number : ");
    UART_Transmiti(n);
    UART_Transmits(" sampling : ");
```

```
  }
  UART_Transmiti(card_is_sampling(n));
}

unsigned char whichTimeBase(void) {
  // Works out which timebase is being used by the currently-selected card.
  // Sends a text representation of the timebase number.
  unsigned char ISAData = readCardStatusRegisterHigh(CardNum);
  int n_timebase = (ISAData >> 5) & 0b11;
  if (verbose) {
    UART_Transmits("Card Number : ");
    UART_Transmiti(CardNum);
    UART_Transmits(" is using timebase ");
  }
  UART_Transmiti(n_timebase);
  return n_timebase;
}

int getBufferPointer (void) {
  // Gets the value of the poiner to the oldest word in the circular buffer.
  int buf;
  unsigned char ISAData = readCardStatusRegisterHigh(CardNum);
  buf = (ISAData & 0b11111) << 8;
  ISAData = readCardStatusRegisterLow(CardNum);
  buf = buf | ISAData;
  return buf;
}

void printBufferPointer(void) {
  // Returns a text representation of the pointer to the oldest word
  // in the circular buffer.
  if (verbose) {
    UART_Transmits("Card Number : ");
    UART_Transmiti(CardNum);
    UART_Transmits(" buffer pointer is 0x");
  }
  UART_Transmitlong(getBufferPointer());
}


// ---------------- Trigger Unit Control ------------------

void writeTriggerUnitControlRegister(unsigned char ISAData) {
  IOwrite(eeprom_read_word(&TimeBaseAddress), ISAData);
  return;
}

void arm_box(void) {
  // Start the box sampling by writing the arm bit to the register.
  if (UART_Receivei()) {
    writeTriggerUnitControlRegister( 0x20 );
    if (verbose) UART_Transmits("All Trigger Units Armed.");
  }
}

void trigger_box(void) {
  // Stop the box sampling (sometime soon) by writing the trigger bit.
  if (UART_Receivei()) {
    writeTriggerUnitControlRegister( 0x10 );
    if (verbose) UART_Transmits("All Trigger Units Triggered.");
  }
}

// ------------------- Time Base Status ---------------------

unsigned char readTimeStatusRegisterHigh(unsigned char n) {
  if (n <= 0) n = 1;
  if (n > 3) n = 3;
```

```
  return IOread(eeprom_read_word(&TimeBaseAddress) + 1 + 2 * n);
}

unsigned char readTimeStatusRegisterLow(unsigned char n) {
  if (n <= 0) n = 1;
  if (n > 3) n = 3;
  return IOread(eeprom_read_word(&TimeBaseAddress) + 2 * n);
}

unsigned char box_is_present() {
  // Returns 1 if box appears to be present, 0 otherwise.
  unsigned char all_ones = (readTimeStatusRegisterHigh(1) == 0xFF) &&
                           (readTimeStatusRegisterLow(1)  == 0xFF) &&
                           (readTimeStatusRegisterHigh(2) == 0xFF) &&
                           (readTimeStatusRegisterLow(2)  == 0xFF) &&
                           (readTimeStatusRegisterHigh(3) == 0xFF) &&
                           (readTimeStatusRegisterLow(3)  == 0xFF);
  return !all_ones;
}

void preTriggerDelay(unsigned char n) {
  // Sends a text representation of the number of pretrigger samples
  // for timebase n.
  if (n <= 0) n = 1;
  if (n > 3) n = 3;
  unsigned char ISAData = readTimeStatusRegisterHigh(n);
  if (verbose) {
    UART_Transmits("Trigger Unit ");
    UART_Transmiti(n);
    UART_Transmits(" Pre-trigger samples is ");
  }
  UART_Transmiti((ISAData >> 4) & 0b111);
  UART_Transmiti(ISAData & 0b1111);
  UART_Transmits("92");
}

void bufferSizeSelect(unsigned char n) {
  // Sends a text representation of the selected buffer size for timebase n.
  if (n <= 0) n = 1;
  if (n > 3) n = 3;
  unsigned char ISAData = readTimeStatusRegisterLow(n);
  if (verbose) {
    UART_Transmits("Time Base Unit ");
    UART_Transmiti(n);
    UART_Transmits(" buffer size is ");
  }
  switch (ISAData >> 6) {
    case 0 : UART_Transmits("A"); break;
    case 1 : UART_Transmits("2"); break;
    case 2 : UART_Transmits("8"); break;
    case 3 : UART_Transmits("4"); break;
    default : UART_Transmits("X"); break;
  } // end switch
}

unsigned char whichTriggerUnit(unsigned char n) {
  // Send a text representation of the trigger-unit that is being used
  // by the timebase n.
  unsigned char trigger_unit;
  if (n <= 0) n = 1;
  if (n > 3) n = 3;
  unsigned char ISAData = readTimeStatusRegisterLow(n);
  if (verbose) {
    UART_Transmits("Time Base Unit ");
    UART_Transmiti(n);
    UART_Transmits(" is using trigger unit ");
  }
  switch ((ISAData >> 4) & 0b11) {
```

```
    // Note the odd mapping; it's not straight through.
    case 0 : trigger_unit = 1; break;
    case 1 : trigger_unit = 2; break;
    case 3 : trigger_unit = 3; break;
    default : trigger_unit = 0; break; // should never see this case
  }
  UART_Transmiti(trigger_unit);
  return trigger_unit;
}

void samplePeriod(unsigned char n) {
  // Sends a text representation of the sample period (in us)
  // for the timebase n.
  if (n <= 0) n = 1;
  if (n > 3) n = 3;
  unsigned char ISAData = readTimeStatusRegisterLow(n);
  if (verbose) {
    UART_Transmits("Trigger Unit ");
    UART_Transmiti(n);
    UART_Transmits(" sampling period is (us) ");
  }
  switch (ISAData & 0b111) {
    case 1 : UART_Transmits("50"); break;
    case 2 : UART_Transmits("20"); break;
    case 3 : UART_Transmits("10"); break;
    case 4 : UART_Transmits("05"); break;
    case 5 : UART_Transmits("02"); break;
    case 6 : UART_Transmits("01"); break;
    default : UART_Transmits("XX"); break;
  }
}

void timeBaseMultiplier(unsigned char n) {
  // Sends a text representation of the sample-period multiplier
  // for timebase n.
  if (n <= 0) n = 1;
  if (n > 3) n = 3;
  unsigned char ISAData = readTimeStatusRegisterLow(n);
  if (verbose) {
    UART_Transmits("Trigger Unit ");
    UART_Transmiti(n);
    UART_Transmits(" time base multiplier is  ");
  }
  switch ((ISAData >> 3) & 0b1) {
    case 0 :
      UART_Transmits("0"); break;  //x1 multiplier
    case 1 :
      UART_Transmits("1");         //x100 multiplier
      if (verbose) {
        UART_Transmits(" x100");
      }
      break;
    default : UART_Transmits("X"); break;
  }
}

// ------------------ Trigger Unit Status Registers ---------------------------

unsigned char readTriggerStatusRegisterHigh(void) {
  return IOread(eeprom_read_word(&TimeBaseAddress) + 1);
}

unsigned char readTriggerStatusRegisterLow(void) {
  return IOread(eeprom_read_word(&TimeBaseAddress));
}

void report_coupling(unsigned char n) {
  // Sends a text representation of the coupling for trigger unit n.
```

```
  if ( n < 1 ) n = 1;
  if ( n > 3 ) n = 3;
  unsigned char ISAData = readTriggerStatusRegisterLow();
  if (verbose) {
    UART_Transmits("Trigger : ");
    UART_Transmiti(n);
    UART_Transmit(' ');
  }
  if ((ISAData >> (2 * n - 2)) & 0b1) {
    UART_Transmit('D');
  } else {
    UART_Transmit('A');
  }
  if (verbose) {
    UART_Transmits("C Coupling");
  }
}

void report_slope(unsigned char n) {
  // Sends a text representation of the slope for trigger unit n.
  if ( n < 1 ) n = 1;
  if ( n > 3 ) n = 3;
  unsigned char ISAData = readTriggerStatusRegisterLow();
  if (verbose) {
    UART_Transmits("Slope on Trigger : ");
    UART_Transmiti(n);
    UART_Transmits(" ");
  }
  if((ISAData >> (2 * n - 1)) & 0b1) {
    UART_Transmits("R"); // rising
  } else {
    UART_Transmits("F"); // falling
  }
}

void reportSlopeAndCouple(unsigned char n) {
  // Sends a text representation of the raw bits for slope and coupling
  // of trigger unit n.
  if ( n < 1 ) n = 1;
  if ( n > 3 ) n = 3;
  unsigned char ISAData = readTriggerStatusRegisterLow();
  if (verbose) {
    UART_Transmits("Slope and Coupling on Trigger : ");
    UART_Transmiti(n);
    UART_Transmits(" ");
  }
  UART_Transmiti((ISAData >> (2 * n - 2)) & 0b11);
}

void enable_threshold_report(unsigned char n) {
  // Set the flag for suppressing or allowing reporting of k.
  // I suppose that the logic looks a bit backward but
  // it is easier to short-cut the reporting functions if
  // k-reporting is to be suppressed.
  if ( n == 1 ) {
    suppress_k_report = 0;
  } else {
    suppress_k_report = 1;
  }
  if (verbose) {
    UART_Transmits("Enable k-reporting: ");
    UART_Transmiti(!suppress_k_report);
    UART_Transmits(" ");
  } else {
    UART_Transmiti(!suppress_k_report);
  }
  return;
}
```

```
void report_threshold(unsigned char n) {
  // Sends an octal representation of the threshold for trigger unit n.
  //
  // There is a small catch with the databox: it will trigger if we
  // write to the control register to select the requested unit.
  if ( any_cards_sampling() || suppress_k_report ) {
    UART_Transmits("000");  // a null value, so to speak
    return;
  }
  if ( n < 1 ) n = 1;
  if ( n > 3 ) n = 3;
  unsigned char ISAData = (0x04 | n) & 0x0F; // redundant filtering of bits
  writeTriggerUnitControlRegister(ISAData);
  while ( thresholdStatus() ) /* wait */ ;
  ISAData = readTriggerStatusRegisterHigh();
  if (verbose) {
    UART_Transmits("Threshold on Trigger : ");
    UART_Transmiti(n);
    UART_Transmits(" ");
  }
  // OCTAL 000 = -2.5V, 377 = 2.5V
  char i;
  for (i = 2; i >= 0; i--) {
    UART_Transmit(((ISAData >> (3 * i)) & 0b111) + 0x30);
  }
}

void report_threshold_in_BCD(unsigned char n) {
  // Sends a decimal representation of the threshold for trigger unit n.
  //
  // There is a small catch with the databox: it will trigger if we
  // write to the control register to select the requested unit.
  if ( any_cards_sampling() || suppress_k_report ) {
    UART_Transmits("+00");  // a null value, so to speak
    return;
  }
  if ( n < 1 ) n = 1;
  if ( n > 3 ) n = 3;
  unsigned char ISAData = (0x04 | n) & 0x0F;
  writeTriggerUnitControlRegister(ISAData);
  while ( thresholdStatus() ) /* wait */ ;
  ISAData = readTriggerStatusRegisterHigh();
  if (verbose) {
    UART_Transmits("Threshold on Trigger : ");
    UART_Transmiti(n);
    UART_Transmits(" ");
  }
  //-99 = -2.5V, +99 = 2.5V
  if (ISAData > 127) {
    UART_Transmit('+');
    ISAData = ISAData - 128;
  } else {
    UART_Transmit('-');
    ISAData = 127 - ISAData;
  }
  unsigned int value = ISAData * 100 / 128;
  unsigned int high = value / 10;
  UART_Transmiti(high);
  UART_Transmiti(value - high * 10);
}

unsigned char thresholdStatus(void) {
  // Returns 1 if the threshold A->D converter is busy.
  unsigned char ISAData = readTriggerStatusRegisterLow();
  return (ISAData >> 7);
}
```

```
// ---------------------------- System Functions ----------------------------

void reportIOBase(void) {
  // Sends a text representation of the current IO base-address.
  if (verbose) {
    UART_Transmits("I/O BASE ADDRESS : 0x");
  }
  UART_Transmitlong(eeprom_read_word(&IOBaseAddress));
}

void setIOBase(void) {
  // Sets the base-addresses in the IO space by writing them into the EEPROM.
  int temp;
  temp =  UART_Receivei() << 8;
  temp |= UART_Receivei() << 4;
  temp |= UART_Receivei();
  eeprom_write_word(&IOBaseAddress, temp);
  eeprom_write_word(&TimeBaseAddress, (temp - 0x10));
  reportIOBase();
}

void setVerbose(void) {
  // Sets the status of the verbose flag.
  verbose = UART_Receivei();
  if (verbose) {
    UART_Transmits("Verbose : ON");
  } else {
    UART_Transmit('F');
  }
}

void menu(void) {
  const char *menu_text =
#   include "Menu.txt"
  ;
  char c;
  const char *addr;
  addr = menu_text;
  while ( (c = pgm_read_byte(addr++)) ) {
    UART_Transmit(c);
  }
}
```

A.10 Menu.txt

```
PSTR("********************************* M E N U  *********************************\r\n"
"\r\n"
"All uppercase and some lowercase commands require a following integer parameter i\r\n"
"e.g. R1 to reset\r\n"
"For convenience RR, AA, TT, & BB are equivalent to R1, A1, T1, & B1 respectively.\r\n"
"Card commands give a response for the selected card and channel unless otherwise indicated.\r\n"
"\r\n"
"*********************** A/D Card Control ***********************\r\n"
"\r\n"
"'Ri' : < i = 0 - 1 > : 1 = reset & hold ring pointers at location 0x0000\r\n"
"'Ni' : < i = 1 - F > : select card number\r\n"
"'Ci' : < i = 1 - 3 > : select channel number\r\n"
"'Di' : < i = 1 - 3 > : send header + 8k values for card N, channel i\r\n"
"   ** NOTE ** this command(Di) also sets Brief mode(B0) & selects Channel(Ci)\r\n"
"'Oi' : < i = 1 - 3 > : get the latest 15 bit value from channel i as octal\r\n"
"'Vi' : < i = 1 - 3 > : get the latest conversion from channel i as BCD voltage\r\n"
"'f' : transmit the latest full scale data range (volts) (for channel NiCi)\r\n"
"'g' : transmit the latest data coupling (A or D) (for channel NiCi)\r\n"
"\r\n"
"*********************** A/D Card Status ***********************\r\n"
"\r\n"
"'xi' : < i = 1 - F > : check existence of card : 1 = appears to exist , 0 = not present\r\n"
"'ai' : < i = 1 - F > : get sampling status : 1 = still sampling, 0 = hold\r\n"
```

```
"'r' : get ring-buffer pointer to oldest data address : HHHH (hexadecimal) \r\n"
"'t' : get card's timebase selection : 1-3 or 0 = timebase 1 at x4\r\n"
"\r\n"
"********************** Trigger Unit Control **********************\r\n"
"\r\n"
"'Ai' : < i = 0 - 1 > : 1 = arm all trigger units (0 = no effect)\r\n"
"'Ti' : < i = 0 - 1 > : 1 = causes a triggering of Databox (0 = no effect)\r\n"
"\r\n"
"********************** Time Base Status *********************\r\n"
"\r\n"
"'di' : < i = 1 - 3 > get thumbwheel selection : dd92 (decimal)\r\n"
"'bi' : < i = 1 - 3 > get buffer size selection : 2, 4, 8kb, A=Active\r\n"
"'pi' : < i = 1 - 3 > get sample period : 01-50microseconds\r\n"
"'mi' : < i = 1 - 3 > get multiplier : 1 = x100, 0 = x1\r\n"
"'ui' : < i = 1 - 3 > get timebase unit trigger # : (1-3)\r\n"
"\r\n"
"********************** Trigger Unit Status **********************\r\n"
"\r\n"
"'ci' : < i = 1 - 3 > get trigger unit coupling: D = 1 = dc, A = 0 = ac\r\n"
"'si' : < i = 1 - 3 > get slope setting: R = 1 = rising, F = 0 = falling\r\n"
"'ki' : < i = 1 - 3 > get trigger level percent : +/-99\r\n"
"'Li' : < i = 1 - 3 > get trigger level percent : octal\r\n"
"'zi' : < i = 0 - 1 > determine threshold-reporting behaviour: 1=enable, 0=disable\r\n"
"\r\n"
"**************************** System Functions ****************************\r\n"
"\r\n"
"'v' : get port and software version No.\r\n"
"'y' : check existence of box: 1 = appears to be present, 0 = not present\r\n"
"'i' : get programmed i/o base address: default=0x0320\r\n"
"'I' : < i = xxx > permanently change hex i/o base address. e.g I320\r\n"
"'Ui' : < i = 1 - 6 > permanently change RS232 baud rate\r\n"
"      1:2400 2:4800 3:9600 4:38400 5:57600 6:115200\r\n"
"'Bi' : < i = 0 - 1 > set output mode : 0 = brief, 1 = verbose \r\n"
"'h' : print this help menu\r\n"
"\r\n")
```

B Databox Management Program – Serial Version

The following Python program will run on any computer that has the Python interpreter installed, along with a number of common extensions for Python. On a Microsoft Windows computer, you may need to install:

1. ActivePython 2.4 (or later, presumably)
2. ActiveTcl 8.4 (for td_browser.tcl)
3. egenix mx_base
4. numpy (preferred for new work) or Numeric (in case some packages need it)
5. Pmw (Python MegaWidgets for Tkinter, download source from web)
6. PmwContribD_r2_0_2 (Contributed widgets for Pmw, download source from web)
7. pyserial
8. pywin32

On a Linux machine, you probably have python already but you may need to install:

1. tkinter (and friends)
2. numpy or Numeric
3. egenix mx (or whatever packege name is used on your distribution)
4. Pmw + PmwContribD (download source from web)
5. pyserial

B.1 dbox_view_usb.py

```
#! /usr/bin/env python
## \file dbox_view_usb.py
## \brief Manager/viewer for the BCD Databox
## \author Peter Jacobs
## \version 1.0, 20-Oct-2004
## \version 1.1, 29-Jan-2007, configuration file, small UI tweaks
## \version 1.2, 05-Feb-2007, select threshold-reporting behaviour
##

versionString = "1.2"

import sys
import os
import gzip
import Tkinter
```

```
import tkFileDialog
import tkSimpleDialog
import tkMessageBox
import Pmw
try:
    from Numeric import *
except:
    try:
        from numpy import *
    except:
        print "Failed to import either numpy or Numeric"
from dbox_services_usb import *
from dbox_data_usb import *
from my_progress import myProgressDialog

# Global data
global dbox_info, dbox_service, dbox_lock

print "Initialize..."
dbox_service = BCDDataBox()
dbox_info = DataBoxInfo(dbox_service)

#------------------------------------------------------------------------------

def setup_GUI_menus(root):
    "Establish the menu options and their function calls"
    menuBar = Tkinter.Menu(root)

    fileMenu = Tkinter.Menu(menuBar)
    fileMenu.add_command(label="Select data directory",
                         command=select_data_dir_dialog)
    fileMenu.add_command(label="Select backup directory",
                         command=select_backup_data_dir_dialog)
    fileMenu.add_separator()
    fileMenu.add_command(label="Read signal config",
                         command=read_config_file_dialog)
    fileMenu.add_command(label="Save signal config",
                         command=save_config_file_dialog)
    fileMenu.add_separator()
    fileMenu.add_command(label="Read run description",
                         command=read_run_description_dialog)
    fileMenu.add_command(label="Save run description",
                         command=save_run_description_dialog)
    fileMenu.add_separator()
    fileMenu.add_command(label="Read shot data from disc", command=import_shot_dialog)
    fileMenu.add_command(label="Save data", command=save_data_dialog)
    fileMenu.add_command(label="Export data to CSV file", command=export_csv_dialog)
    fileMenu.add_separator()
    fileMenu.add_command(label="Save all plots (as postscript)",
                         command=save_postscript_allplot_dialog_askName)
    fileMenu.add_command(label="Save plots on screen (as postscript)",
                         command=save_postscript_plot_dialog_askName)
    fileMenu.add_command(label="Save plots on screen as shot-N.ps",
                         command=save_postscript_plot_dialog_automatic)
    fileMenu.add_separator()
    fileMenu.add_command(label="Quit", command=quit_dialog)
    menuBar.add_cascade(label="File", menu=fileMenu)

    root.config(menu=menuBar)
    return

def quit_dialog():
    global root, dataLoadedButNotSaved
    if dataLoadedButNotSaved:
        dialog = Pmw.MessageDialog(root, title="Quit Databox Manager",
                     defaultbutton=0, buttons=("Quit anyway", "Oops"),
                     message_text="You have loaded data which has not been saved.")
        result = dialog.activate()
```

```
    else:
        result = "Quit anyway"
    if result == "Quit anyway":
        sys.exit()
    return

def read_config_file_dialog(fileName = None):
    """
    Offer the option of selecting a configuration file.

    """
    global dbox_info, dbox_service, signalConfigTextWidget
    if fileName == None:
        fileName = tkFileDialog.askopenfilename()
    if fileName:
        f = open(fileName, "r")
        dbox_info.signalConfigText = f.read()
        f.close
        T = signalConfigTextWidget
        T.delete('1.0', Tkinter.END)
        T.insert(Tkinter.END, dbox_info.signalConfigText)
        scan_signal_config(dbox_info, dbox_service)
        print "Finished reading config file."
    return

def save_config_file_dialog():
    """
    Offer the option of saving a configuration file.

    """
    global dbox_info, dbox_service, signalConfigTextWidget
    fileName = tkFileDialog.asksaveasfilename()
    if fileName:
        T = signalConfigTextWidget
        dbox_info.signalConfigText = T.get('1.0',Tkinter.END)
        scan_signal_config(dbox_info, dbox_service)
        f = open(fileName, "wt")
        f.write(dbox_info.signalConfigText)
        f.close
        print "Finished writing config file."
    return

def read_run_description_dialog(fileName = None):
    """
    Look for an existing run description to import.
    """
    global dbox_info, runDescTextWidget
    if fileName == None:
        fileName = tkFileDialog.askopenfilename()
    if fileName:
        f = open(fileName, "r")
        dbox_info.runDescription = f.read()
        f.close()
        T = runDescTextWidget
        T.delete('1.0', Tkinter.END)
        T.insert(Tkinter.END, dbox_info.runDescription)
    return

def save_run_description_dialog():
    """
    Offer the option of saving the run description text to a file.
    """
    global dbox_info, runDescTextWidget
    fileName = tkFileDialog.asksaveasfilename()
    if fileName:
        T = runDescTextWidget
        dbox_info.runDescText = T.get('1.0',Tkinter.END)
        f = open(fileName, "wt")
```

```
        f.write(dbox_info.runDescText)
        f.close
    return

def save_postscript_allplot_dialog_askName():
    """
    Offer the option of saving all plots to a file in postscript format.
    """
    global dbox_info
    fileName = tkFileDialog.asksaveasfilename()
    if fileName:
        save_postscript_plot(fileName, 1)
    return

def save_postscript_plot_dialog_askName():
    """
    Offer the option of saving the screen view to a file in postscript format.
    """
    global dbox_info
    fileName = tkFileDialog.asksaveasfilename()
    if fileName:
        save_postscript_plot(fileName, 0)
    return

def save_postscript_plot_dialog_automatic():
    """
    Save the plot to an automatically-named file in postscript format.
    """
    global dbox_info
    # Try to build a unique file name for the plot.
    shotId = dbox_info.shotId
    N = 1
    fileName = shotId + "-"+ str(N) + ".ps"
    while os.path.exists(fileName):
        N = N + 1
        fileName = shotId + "-"+ str(N) + ".ps"
    if fileName:
        save_postscript_plot(fileName, 0)
    return

def save_postscript_plot(fileName, allPlots=0):
    """
    Do the actual work of saving the plot in postscript format.
    """
    global canvasWidget, canvasWidth, canvasHeight
    c = canvasWidget
    if allPlots:
        # Render all plots to postscript.
        try:
            x1, y1, x2, y2 = c.bbox("boxes")
        except:
            # in case there are no items found on the canvas
            x1 = 0; x2=canvasWidth; y1=0; y2=canvasHeight
        print "bbox: x1=", x1, "y1=", y1, "x2=", x2, "y2=", y2
        width = x2 - x1
        height = y2 - y1
        if height > 2 * width:
            c.postscript(file=fileName, y=y1, height=height, pageheight='24c')
        else:
            c.postscript(file=fileName, y=y1, height=height, pagewidth='12c')
    else:
        # Render just the plots that are presently seen on screen.
        c.postscript(file=fileName, pagewidth='12c')
    return

def arm_databox_dialog():
    """
    After asking for confirmation
```

```
    (because it will destroy any data in the databox buffers),
    do synchronisation on all used cards and arm (i.e. start sampling)
    """
    global root
    dialog = Pmw.MessageDialog(root, title="Preparing to arm databox",
             defaultbutton=0, buttons=("Arm Databox", "Cancel"),
             message_text="Beware that arming will wipe data in databox buffers.")
    result = dialog.activate()
    if result == "Arm Databox":
        dbox_info.wait_to_acquire_lock()
        dbox_service.reset_cards_with_list_and_arm(dbox_info.cardDict.keys())
        dbox_info.release_lock()
    return

def trigger_databox_dialog():
    "Send trigger signal to databox without asking for confirmation."
    dbox_info.wait_to_acquire_lock()
    dbox_service.trigger()
    dbox_info.release_lock()
    return

def select_data_dir_dialog():
    "Bring up a GUI dialog for directory selection."
    global dbox_info, dataDirEntryWidget
    dataDir = tkFileDialog.askdirectory()
    if dataDir:
        dbox_info.dataDir = dataDir
        e = dataDirEntryWidget
        e.delete(0, Tkinter.END)
        e.insert(0, dataDir)
    return

def select_backup_data_dir_dialog():
    "Bring up a GUI dialog for directory selection."
    global dbox_info, backupDataDirEntryWidget
    dataDir = tkFileDialog.askdirectory()
    if dataDir:
        dbox_info.backupDataDir = dataDir
        e = backupDataDirEntryWidget
        e.delete(0, Tkinter.END)
        e.insert(0, dataDir)
    return

#----------------------------------------------------------------------

def do_collect_data():
    """
    Fetch the data from the databox buffers for the actively-used cards.

    If a card is still sampling when we get to it, wait a little and
    check again every so often.
    """
    global root, dbox_info, dbox_service, dataLoadedButNotSaved
    dbox_info.wait_to_acquire_lock()
    # Do a check of all of the configured cards
    # to see if they have stopped sampling.
    cardList = dbox_info.cardDict.keys()
    ncss = 0
    for card_id in cardList:
        card = dbox_info.cardDict[card_id]
        print "Collect data for card:", card.id_number
        card_status = dbox_service.get_card_status(card_id)
        ncss = card_status["sampling"]
    if ncss > 0:
        print "Cards still sampling; collect failed."
        dialog = Pmw.MessageDialog(root, title="Collect data failed",
                    defaultbutton=0, buttons=("OK", "This is Crap"),
                    message_text="You have tried to collect the sampled data\n" +
```

```
                                   "before all of the cards have finished sampling.")
        result = dialog.activate()
        root.update()
        wait_a_while(200) # seem to need a bit of time to clear dialog
        print "Ignored request to collect data."
        dbox_info.release_lock()
        return
    else:
        # Do the collection.
        busy()
        timebase_status = dbox_service.get_timebase_status(1)
        pd = myProgressDialog(maxValue=len(cardList), title="dbox_view",
                              message="Collecting data from the A/D cards.")
        for card_id in cardList:
            root.update()
            card = dbox_info.cardDict[card_id]
            print "Collect data for card:", card.id_number
            card_status = dbox_service.get_card_status(card_id)
            card_dt = float(timebase_status[card_status["time_base"]]["sample_period"]) / \
                      float(card_status["time_scale"])
            buffer_size = timebase_status[card_status["time_base"]]["buffer_size"]
            for channel_id in card.channels_used.keys():
                mux = card.subchannels[channel_id]
                print "Channel:", channel_id, "SubChannel count:", mux
                attempt_count = 0
                text_data = None
                while text_data == None and attempt_count < 3:
                    text_data = dbox_service.read_data_buffer(card_id, channel_id)
                    attempt_count += 1
                if text_data == None:
                    print "Failed to collect data in three attempts."
                    print "Proceeding to the next channel."
                    continue
                voltage_demux, ground_input, FS = \
                    dbox_service.decode_data_buffer(text_data, 0, buffer_size, mux, 1)
                if mux < 1:
                    # No multiplexing
                    subchannel_id = 0
                    signalId = (card_id, channel_id, subchannel_id)
                    if not dbox_info.signalDict.has_key(signalId): continue
                    try:
                        signal = dbox_info.signalDict[signalId]
                        signal.dt = card_dt
                        signal.raw_data = voltage_demux[0]
                        signal.FS = FS
                        print "Signal", signalId, "dt_sample=", signal.dt, \
                              "number-of-samples=", len(signal.raw_data)
                        scale_signal_data(signal)
                    except Exception:
                        print "Problems trying to collect signal", signalId
                else:
                    # Multiplexed signals are labelled 1..mux
                    subchannel_list = range(1,mux+1)
                    for subchannel_id in subchannel_list:
                        signalId = (card_id, channel_id, subchannel_id)
                        try:
                            signal = dbox_info.signalDict[signalId]
                            signal.dt = card_dt * len(subchannel_list)
                            signal.raw_data = voltage_demux[subchannel_id - 1]
                            signal.FS = FS
                            print "Signal", signalId, "dt_sample=", signal.dt, \
                                  "number-of-samples=", len(signal.raw_data)
                            scale_signal_data(signal)
                        except Exception:
                            print "Problems trying to collect signal", signalId
                    # end of loop body for multiplexed signals for one card
                # end of loop body for collecting data for one channel
            pd.progress += 1
```

```
            # end of loop body for collecting data for one card
        pd.close()
        not_busy()
        dataLoadedButNotSaved = 1
    # We are done.
    dbox_info.release_lock()
    print "Begin refreshing plot page."
    refresh_plot_page()
    print "Done refreshing plot page."
    # Rainer wants the plot page to be brought up straight after collecting the data.
    global nb
    nb.selectpage("Plotted Data")
    return

def import_shot_dialog():
    "Load previously recorded data from file."
    # load run description
    # load signals
    # load data
    # put this data into the GUI widgets
    global root, dbox_info
    global signalConfigTextWidget
    global shotIdEntryWidget, dataDirEntryWidget
    shotDir = tkFileDialog.askdirectory(title="Choose a shot directory")
    if shotDir:
        busy()
        print "Import shot description and data from %s" % shotDir
        (dataDir, shotId) = os.path.split(shotDir)
        print "shotId=", shotId, "dirName=", shotDir
        e = dataDirEntryWidget
        e.delete(0, Tkinter.END)
        e.insert(0, dataDir)
        dbox_info.dataDir = dataDir
        e = shotIdEntryWidget
        e.delete(0, Tkinter.END)
        e.insert(0, shotId)
        dbox_info.shotId = shotId
        fileName = os.path.join(shotDir, shotId + ".txt")
        if os.path.exists(fileName):
            print "Import run description from", fileName
            read_run_description_dialog(fileName)
        fileName = os.path.join(shotDir, shotId + ".config")
        if os.path.exists(fileName):
            print "Import signal configuration from", fileName
            read_config_file_dialog(fileName)
        else:
            noConfigDialog = Pmw.MessageDialog(root, title="Importing data.",
                                   defaultbutton=0, buttons=("OK",),
                                   message_text="There is no signal config file.")
            result = noConfigDialog.show()
            not_busy()
            root.update()
            wait_a_while(200) # seem to need a bit of time to clear dialog
            return
        pd = myProgressDialog(maxValue=len(dbox_info.signalDict.keys()),
                              title="dbox_view",
                              message="Importing data from files.")
        for signalId in dbox_info.signalDict.keys():
            root.update()
            signal = dbox_info.signalDict[signalId]
            extn = str(signalId[0]) + str(signalId[1]) + str(signalId[2])
            fileName = shotId + "A." + extn + ".gz"
            print "Reading data file for signal", signalId, "fileName", fileName
            try:
                fdata = gzip.open(os.path.join(shotDir, fileName), "rb")
                read_data_from_file(dbox_info, signal, fdata, extn)
                fdata.close
                reconstruct_raw_voltages(signal)
```

```
            except:
                print "Failed to read data file:", fileName
            pd.progress += 1
        pd.close()
        not_busy()
        print "Begin refreshing plot page."
        refresh_plot_page()
        print "Done refreshing plot page."
    print "Finished importing data."
    dataLoadedButNotSaved = 0  # memory copy is same as disc copy
    global nb
    nb.selectpage("Plotted Data")
    return

def apply_current_scales_to_data():
    """
    Assuming that the on-screen signal configuration has been updated,
    rescale the in-memory raw data to get new scaled data.
    """
    global dbox_info, dataLoadedButNotSaved, signalConfigTextWidget
    dbox_info.signalConfigText = signalConfigTextWidget.get('1.0',Tkinter.END)
    rescan_signal_config_for_new_scales(dbox_info)
    for signalId in dbox_info.signalDict.keys():
        signal = dbox_info.signalDict[signalId]
        scale_signal_data(signal)
    dataLoadedButNotSaved = 1
    return

def save_data_control_s(event):
    """
    User pressed Control-s.
    """
    print "Activating save_data_dialog procedure."
    save_data_dialog()
    return

def save_data_dialog():
    """
    Write out all of the in-memory data,
    making sure that we have the latest information from
    the GUI display widgets.
    """
    global root, dbox_info, dataLoadedButNotSaved
    global runDescTextWidget, signalConfigTextWidget
    setShotIdFromEntryWidget(None)
    setDataDirFromEntryWidget(None)
    setBackupDataDirFromEntryWidget(None)
    dbox_info.runDescription = runDescTextWidget.get('1.0',Tkinter.END)
    dbox_info.signalConfigText = signalConfigTextWidget.get('1.0',Tkinter.END)
    targetDir = os.path.join(dbox_info.dataDir, dbox_info.shotId)
    if os.path.exists(targetDir):
        dialog = Pmw.MessageDialog(root, title="Save Data",
                    defaultbutton=0, buttons=("Proceed", "Cancel"),
                    message_text="You are about to overwrite existing data files.")
        result = dialog.activate()
        root.update()
        wait_a_while(200) # seem to need a bit of time to clear dialog
    else:
        result = "Proceed"
    if result == "Proceed":
        busy()
        try:
            do_save_all_data(dbox_info)
            dataLoadedButNotSaved = 0
            dialog_message_text = "Finished saving data."
        except:
            dialog_message_text = "Failed to save data."
        not_busy()
```

```
        print dialog_message_text
        dialog = Pmw.MessageDialog(root, title="Saving data.",
                    defaultbutton=0, buttons=("OK",),
                    message_text=dialog_message_text)
        result = dialog.show()
        root.update()
        wait_a_while(200) # seem to need a bit of time to clear dialog
    return


def export_csv_dialog():
    """
    Write out all of the in-memory data,
    making sure that we have the latest information from
    the GUI display widgets.
    """
    global root, dbox_info
    global runDescTextWidget, signalConfigTextWidget
    setShotIdFromEntryWidget(None)
    setDataDirFromEntryWidget(None)
    setBackupDataDirFromEntryWidget(None)
    dbox_info.runDescription = runDescTextWidget.get('1.0',Tkinter.END)
    dbox_info.signalConfigText = signalConfigTextWidget.get('1.0',Tkinter.END)
    targetDir = os.path.join(dbox_info.dataDir, dbox_info.shotId)
    busy()
    do_export_csv_data(dbox_info)
    not_busy()
    dialog = Pmw.MessageDialog(root, title="Export data to CSV.",
                               defaultbutton=0, buttons=("OK",),
                               message_text="Finished exporting data.")
    result = dialog.show()
    root.update()
    wait_a_while(200) # seem to need a bit of time to clear dialog
    return

#----------------------------------------------------------------------

def setShotIdFromEntryWidget(event):
    global dbox_info, shotIdEntryWidget
    dbox_info.shotId = shotIdEntryWidget.get()
    print "Shot identity now set to ", dbox_info.shotId
    return

def setDataDirFromEntryWidget(event):
    global dbox_info, dataDirEntryWidget
    dbox_info.dataDir = dataDirEntryWidget.get()
    print "Data directory now set to ", dbox_info.dataDir
    return

def setBackupDataDirFromEntryWidget(event):
    global dbox_info, backupDataDirEntryWidget
    dbox_info.backupDataDir = backupDataDirEntryWidget.get()
    print "Backup data directory now set to ", dbox_info.backupDataDir
    return

def layout_run_description_page(parent):
    "Layout the display of the data location and shot identity."
    global dbox_info, shotIdEntryWidget, runDescTextWidget
    global backupDataDirEntryWidget, dataDirEntryWidget
    f = Tkinter.Frame(parent, borderwidth=1)
    #
    row = 0
    l = Tkinter.Label(f, text="Shot Identity:")
    l.grid(row=row, column=0, sticky='w')
    e = Tkinter.Entry(f, bg="white", relief='sunken', width=70)
    e.insert(0, dbox_info.shotId)
    e.grid(row=row, column=1, sticky='w')
    e.bind('<Return>', setShotIdFromEntryWidget)
```

```
    shotIdEntryWidget = e
    #
    row = 1
    l = Tkinter.Label(f, text="Data Directory:")
    l.grid(row=row, column=0, sticky='w')
    e = Tkinter.Entry(f, bg="white", relief='sunken', width=70)
    e.insert(0, dbox_info.dataDir)
    e.grid(row=row, column=1, sticky='w')
    e.bind('<Return>', setDataDirFromEntryWidget)
    dataDirEntryWidget = e
    #
    row = 2
    l = Tkinter.Label(f, text="Backup Directory:")
    l.grid(row=row, column=0, sticky='w')
    e = Tkinter.Entry(f, bg="white", relief='sunken', width=70)
    e.insert(0, dbox_info.backupDataDir)
    e.grid(row=row, column=1, sticky='w')
    e.bind('<Return>', setBackupDataDirFromEntryWidget)
    backupDataDirEntryWidget = e
    #
    # Display the run description in a scrollable text widget
    row = 3
    l = Tkinter.Label(f, text="Run Description:")
    l.grid(row=row, column=0, sticky='nw')
    twf = Tkinter.Frame(f)
    s = Tkinter.Scrollbar(twf)
    T = Tkinter.Text(twf, bg="white")
    s.pack(side='right', fill='both')
    T.pack(side='left', fill='both')
    s.config(command=T.yview)
    T.config(yscrollcommand=s.set)
    T.insert(Tkinter.END, dbox_info.runDescription)
    twf.grid(row=row, column=1)
    runDescTextWidget = T
    #
    f.pack(expand=1, fill='both')
    return

def layout_signal_config_page(parent):
    "Layout the display of the signal configuration file."
    global dbox_info, signalConfigTextWidget
    f = Tkinter.Frame(parent, borderwidth=1)
    #
    # Display the run description in a scrollable text widget
    row = 0
    l = Tkinter.Label(f, text="Signals:")
    l.grid(row=row, column=0, sticky='nw')
    twf = Tkinter.Frame(f)
    s = Tkinter.Scrollbar(twf)
    T = Tkinter.Text(twf, bg="white")
    s.pack(side='right', fill='both')
    T.pack(side='left', fill='both')
    s.config(command=T.yview)
    T.config(yscrollcommand=s.set)
    T.insert(Tkinter.END, dbox_info.signalConfigText)
    twf.grid(row=row, column=1)
    signalConfigTextWidget = T
    #
    rescaleB = Tkinter.Button(f, text="Rescale in-memory data",
                              command=apply_current_scales_to_data)
    rescaleB.grid(row=1,column=0)
    f.pack(expand=1, fill='both')
    return

#------------------------------------------------------------------------

def layout_databox_status_page(parent):
    "Layout the display of the databox status information."
```

```
global dbox_info
global cardStatusEntryWidget
global timebaseStatusEntryWidget
global triggerStatusEntryWidget
global stateOfPlayLabel
#
# Put the whole collection into a frame that doesn't expand
# so that we don't end up with all the entries spread out.
gStatus = Pmw.Group(parent, tag_text="Status")
gStatus.pack()

# Right side of status group will contain trigger units and timebase units
fRight = Tkinter.Frame(gStatus.interior(), borderwidth=1)
fRight.pack(side='right', expand=1, fill='both')
#
g1 = Pmw.Group(fRight, tag_text="Trigger Units")
g1.pack(expand=1, fill='both')
triggerStatusEntryWidget = {}
for i in range(3):
    l = Tkinter.Label(g1.interior(), text=("%d:" % (i+1)) )
    l.grid(row=i, column=0, sticky='w')
    e = Tkinter.Entry(g1.interior(), relief='sunken', width=50)
    e.grid(row=i, column=1, sticky='w')
    triggerStatusEntryWidget[i+1] = e
#
g2 = Pmw.Group(fRight, tag_text="Timebase Units")
g2.pack(expand=1, fill='both')
timebaseStatusEntryWidget = {}
for i in range(3):
    l = Tkinter.Label(g2.interior(), text=("%d:" % (i+1)) )
    l.grid(row=i, column=0, sticky='w')
    e = Tkinter.Entry(g2.interior(), relief='sunken', width=50)
    e.grid(row=i, column=1, sticky='w')
    timebaseStatusEntryWidget[i+1] = e
#
# Left side of status group will contain status of cards
fLeft = Tkinter.Frame(gStatus.interior(), borderwidth=1)
fLeft.pack(side='left', expand=1, fill='both')
g3 = Pmw.Group(fLeft, tag_text="Cards")
g3.pack(expand=1, fill='both')
cardStatusEntryWidget = {}
for i in range(7):
    l = Tkinter.Label(g3.interior(), text=("%d:" % (i+1)) )
    l.grid(row=i, column=0, sticky='w')
    e = Tkinter.Entry(g3.interior(), relief='sunken', width=50)
    e.grid(row=i, column=1, sticky='w')
    if dbox_service.card_is_present(i+1):
        e.configure(bg='gray')
    cardStatusEntryWidget[i+1] = e
#
gAction = Pmw.Group(parent, tag_text="Action")
gAction.pack()
refreshB = Tkinter.Button(gAction.interior(), text="Refresh status",
                          command=refresh_databox_status_page)
armB = Tkinter.Button(gAction.interior(), text="Arm databox",
                      command=arm_databox_dialog)
triggerB = Tkinter.Button(gAction.interior(), text="Send trigger",
                          command=trigger_databox_dialog)
collectB = Tkinter.Button(gAction.interior(), text="Collect data",
                          command=do_collect_data)
refreshB.pack(side='left')
armB.pack(side='left')
triggerB.pack(side='left')
collectB.pack(side='left')
if not dbox_info.databoxIsPresent:
    refreshB.configure(state="disabled")
    armB.configure(state="disabled")
    triggerB.configure(state="disabled")
```

```
        collectB.configure(state="disabled")
    #
    return

def refresh_databox_status_page(rescheduleCheckWhenDone=0):
    """
    Do status checking without upsetting any other activities
    of the databox.
    """
    global dbox_info
    global cardStatusEntryWidget
    global timebaseStatusEntryWidget
    global triggerStatusEntryWidget
    global stateOfPlayLabel
    #
    if not dbox_info.databoxIsPresent:
        print "Cannot refresh databox status page if the databox is not present."
        return

    if dbox_info.lock_is_free():
        dbox_info.wait_to_acquire_lock()
        for i in range(1,4):
            triggerStatusEntryWidget[i].delete(0, Tkinter.END)
        for i in range(1,4):
            timebaseStatusEntryWidget[i].delete(0, Tkinter.END)
        for i in range(1,8):
            cardStatusEntryWidget[i].delete(0, Tkinter.END)
        #
        for i in range(1,4):
            trs = dbox_service.get_trigger_unit_settings(i)
            display_data = "slope=%s  coupling=%s  threshold=%6.0f" % \
                           (trs["slope"], trs["coupling"], trs["threshold"])
            triggerStatusEntryWidget[i].insert(Tkinter.END, display_data)
        #
        tbs = dbox_service.get_timebase_status()
        for i in range(1,4):
            display_data = "dt=%6dus  pretrigger=%4d  trigger_unit=%d  buffer_size=%d" \
                           % (tbs[i]['sample_period'], tbs[i]['pretrigger_samples'],
                              tbs[i]['trigger_unit'], tbs[i]['buffer_size'])
            timebaseStatusEntryWidget[i].insert(Tkinter.END, display_data)

        for card_id in dbox_info.cardDict.keys():
            card = dbox_info.cardDict[card_id]
            cs = dbox_service.get_card_status(card_id)
            display_text = "sampling=%d timebase=%d time_scale=%d first_word=%d" % \
                           (cs["sampling"], cs["time_base"],
                            cs["time_scale"], cs["first_word"])
            ew = cardStatusEntryWidget[card_id]
            ew.insert(Tkinter.END, display_text)
            if cs["sampling"] > 0:
                ew.configure(bg='yellow')
            else:
                ew.configure(bg='gray')
        # We are done updating the status page
        dbox_info.release_lock()
    else:
        # We have been locked out by some other function.
        pass
    #
    if rescheduleCheckWhenDone > 0:
        root.after(1000, refresh_databox_status_page,1)
    return

#------------------------------------------------------------------------

def layout_plot_page(parent):
    "Sets up a scrolled canvas for later drawing of the individual plots."
    global dbox_info, canvasWidget, plotsPerRowComboBox
```

```
    global canvasWidth, canvasHeight, configData
    # Put a drop-down list of options at the top of the page.
    f0 = Tkinter.Frame(parent, borderwidth=1)
    f0.pack(expand=1, fill="both")
    Tkinter.Label(f0, text="Plots per row:").pack(side="left")
    plotsPerRowComboBox = Pmw.ComboBox(f0, scrolledlist_items=("1","2","3","4"),
                                       entry_width=5,
                                       selectioncommand=update_plot_page)
    plotsPerRowComboBox.selectitem(configData['plots-per-row'])
    plotsPerRowComboBox.pack(side="left")
    f = Tkinter.Frame(parent, borderwidth=1)
    f.pack(expand=1, fill="both")
    # We want a large canvas that can be scrolled.
    # Since the Python Megawidgets version seems to be broken,
    # build our own with standard Tkinter widgets.
    scf = Tkinter.Frame(f)
    scf.pack(expand=1, fill="both")
    s = Tkinter.Scrollbar(scf)
    canvasWidth = int(configData['plot-window-width'])
    canvasHeight = int(configData['plot-window-height']) * 2
    # The screen view of the canvas with be shorter than its full height.
    c = Tkinter.Canvas(scf, bg="gray85", width=canvasWidth,
                       height=int(configData['plot-window-height']),
                       scrollregion=(0,0,canvasWidth,canvasHeight) )
    s.pack(side='right', fill='y')
    c.pack(side='left', fill='both')
    s.config(command=c.yview)
    c.config(yscrollcommand=s.set)
    canvasWidget = c
    return

def plot_a_signal(signal, xleft, ytop, xsize, ysize):
    "Plot the raw voltages."
    global canvasWidget
    label = "%s (%d %d %d)" % (signal.name, signal.card_id, signal.channel_id,
                               signal.subchannel_id)
    c = canvasWidget
    # outline box
    c.create_rectangle(xleft, ytop, xleft+xsize, ytop+ysize, tags="boxes")
    # signal label at top middle
    ymid = ytop + ysize
    xmid = xleft + xsize / 2
    c.create_text(xmid, ytop+10, text=label, tags="text")
    # axes
    x1 = xleft + int(0.10 * xsize)
    x2 = xleft + int(0.90 * xsize)
    xaxis_length = x2 - x1
    y0 = ytop + int(0.50 * ysize)
    yaxis_length = int(0.40 * ysize)
    c.create_line(x1, y0-yaxis_length, x1, y0+yaxis_length, tags="axes")
    c.create_line(x1, y0, x2, y0, tags="axes")
    c.create_text(x1, y0-yaxis_length, text=("%.1f" % signal.FS),
                  anchor='e', tags="axes")
    c.create_text(x1, y0+yaxis_length, text=("%.1f" % -signal.FS),
                  anchor='e', tags="axes")
    try:
        N = len(signal.raw_data)
    except:
        N = 0
    tmax = int(signal.dt * N)
    c.create_text(x2, y0, text=("%d" % tmax), anchor='n', tags="axes")
    if N > 0:
        # now, plot every nth data point
        nth = 1
        x = x1 + (xaxis_length * arange(0, N, nth)) / N
        y = y0 - (signal.raw_data[0::nth] / signal.FS * yaxis_length)
        coordList = transpose(array([x,y])).tolist()
        c.create_line(coordList, tags="datalines", fill="red")
```

```
    return

def update_plot_page(arg):
    "Callback function for the ComboBox."
    global dbox_info
    if len(dbox_info.signalDict.keys()) > 0:
        refresh_plot_page()
    return

def refresh_plot_page():
    "Draw all of the signals onto the plot page."
    global canvasWidget, plotsPerRowComboBox
    busy()
    c = canvasWidget
    c.delete("boxes")
    c.delete("text")
    c.delete("axes")
    c.delete("datalines")
    plotsPerRow = int(plotsPerRowComboBox.get())
    xsize = int(600 / plotsPerRow)
    ysize = 100
    x0 = 10
    y0 = 0
    plotsToLeft = 0
    ytop = y0
    signalIdList = dbox_info.signalDict.keys()
    signalIdList.sort()
    for i in signalIdList:
        signal = dbox_info.signalDict[i]
        xleft = x0 + xsize * plotsToLeft
        plot_a_signal(signal, xleft, ytop, xsize, ysize)
        plotsToLeft += 1
        if plotsToLeft >= plotsPerRow:
            ytop += ysize
            plotsToLeft = 0
    xmax = x0 + plotsPerRow * xsize
    ymax = ytop + ysize
    c.configure(scrollregion=(0,0,xmax,ymax))
    not_busy()
    return

#---------------------------------------------------------------------------

def wait_a_while(ms):
    """Wait for the specified number of milliseconds.

    This function is called in various places to allow time
    for the GUI elements to be updates.  If there are now
    pauses in some of the processor-intensive procedures,
    the GUI becomes disconcertingly unresponsive.
    """
    global root, wait_flag
    print "Waiting...",
    wait_flag = Tkinter.StringVar()
    root.after(ms, lambda : wait_flag.set("done"))
    root.wait_variable(wait_flag)
    print "done."
    return

def busy():
    "Display the watch cursor to indicate that the program is busy."
    global root
    root.config(cursor="watch")
    root.update()
    return

def not_busy():
    "Go back to the default cursor to indicate that the program is not busy."
```

```
    global root
    root.config(cursor="")
    root.update()
    return

#-------------------------------------------------------------------------------

def initializeConfigData():
    # We will keep the program's configuration data as a dictionary of strings.
    global configData
    configData = {
        'plots-per-row': '1',
        'plot-window-width': '620',
        'plot-window-height': '400',
        'enable-threshold-reporting': '0',
        'data-dir': 'data',
        'backup-data-dir': 'data-backup',
        }
    iniFileName = "dbox_view.ini"
    if os.path.isfile(iniFileName):
        print "Read user configData from", iniFileName
        iniFile = open(iniFileName, "r")
        for line in iniFile:
            # Remove new-line, carriage-return and redundant white-space.
            str = line.replace('\n', '')
            str = line.replace('\r', '')
            while line.find('  ') >= 0:
                line = line.replace('  ', ' ')
            line = line.strip()
            if len(line) == 0: continue # nothing left of the line
            if line[0] == '#': continue # ignore comment line
            itemList = line.split()
            try:
                print "    setting \'%s\' to value \'%s\'" % (itemList[0],itemList[1])
                configData[itemList[0]] = itemList[1]
            except:
                pass
    return

#-------------------------------------------------------------------------------

if __name__ == '__main__':
    global root, wait_flag
    root = Tkinter.Tk()
    Pmw.initialise()
    initializeConfigData()
    dbox_info.dataDir = configData['data-dir']
    dbox_info.backupDataDir = configData['backup-data-dir']
    root.title("BCD DataBox Manager")
    busy()
    Pmw.aboutversion(versionString)
    Pmw.aboutcopyright("Centre for Hypersonics 2004,2007")
    Pmw.aboutcontact("Peter Jacobs\nemail: peterj@mech.uq.edu.au")
    if len(sys.argv) > 1:
        first_cmd_arg = sys.argv[1]
    else:
        first_cmd_arg = ""
    if first_cmd_arg.rfind("v") >= 0:
        # Only advertise if the first command line argument is something
        # like --version -version -v version v ...
        about = Pmw.AboutDialog(root, applicationname="BCD Databox Manager")
        root.lower(about)  # for advertising...
        root.after(3000, lambda : about.lower())

    setup_GUI_menus(root)
    nb = Pmw.NoteBook(root)
    pRun = nb.add("Run Information")
    pSignal = nb.add("Signal Config")
```

```
    pStatus = nb.add("Databox")
    pPlot = nb.add("Plotted Data")
    layout_run_description_page(pRun)
    layout_signal_config_page(pSignal)
    layout_databox_status_page(pStatus)
    layout_plot_page(pPlot)
    nb.pack(fill=Tkinter.BOTH, expand=1)
    nb.setnaturalsize()
    root.bind("<Control-s>", save_data_control_s) # single key-stroke to save data

    dataLoadedButNotSaved = 0
    if dbox_info.databoxIsPresent:
        dbox_service.enable_threshold_reporting(int(configData['enable-threshold-reporting']))
        refresh_databox_status_page(1)
    else:
        dialog = Pmw.MessageDialog(root, title="Databox Status Note.",
            defaultbutton=0, buttons=("OK",),
            message_text="Cannot see the databox.\nIs it plugged in and turned on?")
        result = dialog.show()
        root.update()
    not_busy()

    # Give control to the GUI elements.
    root.mainloop()

#-----------------------------------------------------------------------------
```

B.2 dbox_services_usb.py

```
#! /usr/bin/env python
## \file dbox_services_usb.py
## \brief Service functions for the BCD databox connected via USB.
## \author Peter Jacobs
## \version 1.0 08-Jan-2007
## \version 1.1 19-Jan-2007 version 3.1 firmware, checksum, try several ports
##
## This module provides a collection of service functions that make
## interaction with the BCD databox via Luke Hilliard's USB
## supervisory card.  This is far more convenient than directly reading
## from and writing to the registers via the ISA bus extender.
##

import os, sys, time, serial
from copy import copy
try:
    from Numeric import *
except:
    try:
        from numpy import *
    except:
        print "Failed to import either numpy or Numeric"


#---------------------------------------------------------------------------

def cleanUpString(str):
    "Clean out new-line, carriage-return, extra spaces and null characters."
    str = str.replace('\n', '')
    str = str.replace('\r', '')
    str = str.replace('\x00', '')
    while str.find('  ') >= 0:
        str = str.replace('  ', ' ')
    return str

def rotate_right(c):
    "Rotate bits in a 16-bit word as per Alex Martelli's 2001 message."
    if c & 1:
        return (c >> 1) | 0x8000
```

```
    else:
        return c >> 1

#-------------------------------------------------------------------------

def open_connection(deviceName, baudrate):
    """
    Opens a USB serial connection and attempts to talk to the databox.
    """
    try:
        print "Try serial port", deviceName, "at", baudrate, "baud"
        # We don't want our program to hang if there is no response.
        sp = serial.Serial(port=deviceName, baudrate=baudrate,
                           bytesize=7, parity='O',
                           stopbits=2, timeout=2.5 )
        print "    Successfully opened serial port", sp.portstr
    except:
        sp = None
        print "    Failed to open serial port", deviceName
        return sp
    if databox_is_present(sp):
        print "    Found databox."
        sp.write('B1'); response = sp.readline()
        sp.flushInput()
        # The verbose response to version command may be several lines.
        sp.write('v'); response = sp.readline()
        while response:
            print cleanUpString(response)
            response = sp.readline()
        sp.write('B0'); response = sp.read(1)
        sp.flushInput()
    else:
        print "    Did not find databox."
        sp.close()
        sp = None
    return sp

def close_connection(sp):
    if sp: sp.close()
    return

def databox_is_present(sp):
    """
    Check for presence of databox by sending a simple command
    and checking the response.
    """
    if sp == None: return 0
    sp.flushInput()
    sp.write('B0'); response = sp.read(1) # should be 'F'
    sp.write('y'); response = sp.read(1)
    if len(response) == 0 or response == '0':
        print "There was no response from the databox."
        print "Check that it is turned on and the cable is connected."
        return 0
    else:
        if response.find('1') >= 0:
            print "Databox responds OK."
            sp.flushInput()
            return 1
        else:
            print "Databox responds strangely."
            print "    response was", response, " but should have been 1"
            sp.flushInput()
            return 0

#-------------------------------------------------------------------------

class BCDDataBox:
```

```
"Provides the basic services for dealing with the BCD databox."

def __init__(self):
    """
    Attempts to open a connection and see if the databox is present.
    """
    # The following name will hold the connection to the serial port.
    self.sp = None
    # Assume that USB ports start at COM5.
    if sys.platform == 'win32':
        deviceList = [('COM1:', 115200),
                      ('COM2:', 115200),
                      ('COM3:', 115200),
                      ('COM4:', 115200),
                      ('COM5:', 230400),
                      ('COM6:', 230400),
                      ('COM7:', 230400),
                      ('COM8:', 230400)]
    else:
        deviceList = [('/dev/ttyS0', 115200),
                      ('/dev/ttyS1', 115200),
                      ('/dev/ttyUSB0', 230400),
                      ('/dev/ttyUSB1', 230400)]
    for deviceName,baudrate in deviceList:
        self.sp = open_connection(deviceName, baudrate)
        if self.sp: break
    return

def finalize(self):
    close_connection(self.sp)
    return

def databox_is_present(self):
    return databox_is_present(self.sp)

def reset_cards_with_list_and_arm(self, cardList):
    """
    Synchronise specified cards and arm the box.

    Card list is the list of card numbers that are
    to be synchronised prior to arming the box.
    This is probably the better service to use because the
    Python functions (synchronise_card and arm) are likely
    to be rather slow.
    """
    for card in cardList:
        if card > 7 or card < 1: continue
        self.sp.write('N'+str(card))
    self.sp.write('A1')
    self.sp.flushInput()
    return

def trigger(self):
    """
    Send a trigger signal to all timebases.
    """
    self.sp.write('T1')
    self.sp.flushInput()
    return

def enable_threshold_reporting(self, i):
    """
    Write 'z1' to enable reporting of trigger-level threshold.
          'z0' to suppress reporting of triger-level threshold.

    When reporting is suppressed, dummy values are returned.
    It seems that the T4 databox does not tolerate any fiddling with
    the k-register when the levels are such that a trigger will occur
```

```
        if the box is armed.
        """
        self.sp.write('z'+str(i)); response = self.sp.read(1)
        self.sp.flushInput()
        return

    def get_trigger_unit_settings(self, unit):
        """
        Return a dictionary indicating the status of the trigger unit.
        """
        if unit > 3: unit = 3
        if unit < 1: unit = 1
        coupling = 'Unknown'
        slope = 'Unknown'
        threshold = -1
        self.sp.flushInput()
        self.sp.write('c'+str(unit)); response = self.sp.read(1)
        if response == 'D': coupling = 'DC'
        if response == 'A': coupling = 'AC'
        self.sp.write('s'+str(unit)); response = self.sp.read(1)
        if response == 'R': slope = '+'
        if response == 'F': slope = '-'
        self.sp.write('k'+str(unit)); response = self.sp.read(3)
        # The following calculation converts from the range -99..99
        # to something closer to the 5..995 range on the dial.
        threshold = (int(cleanUpString(response)) + 99)*5
        self.sp.flushInput()
        return {"threshold": threshold, "slope": slope, "coupling": coupling}

    def card_is_present(self, ncard):
        """
        Returns 1 is a card is present.

        The test is to assume that for an existing card,
        the latest 15-bit value will be something other
        than 77777(octal). If a card is not present, the
        resistor networks will pull the bus high.
        """
        if self.sp == None: return 0
        self.sp.flushInput()
        self.sp.write('x'+str(ncard)); response = self.sp.read(1)
        self.sp.flushInput()
        return response == '1'

    def card_is_still_sampling(self, ncard):
        """
        Returns 1 if the card is still sampling, 0 otherwise.
        """
        self.sp.flushInput()
        self.sp.write('a'+str(ncard)); response = self.sp.read(1)
        self.sp.flushInput()
        if response == 'T' or response == '1':
            return 1
        else:
            return 0

    def still_sampling(self, cardList):
        """
        Returns True if at least one card in the supplied list
        is still sampling.
        """
        count = 0
        for id in cardList:
            if self.card_is_still_sampling(id): count += 1
        return count > 0

    def get_card_status(self, ncard):
        """
```

```
        Return a dictionary containing the card status.
        """
        sampling = self.card_is_still_sampling(ncard)
        self.sp.flushInput()
        self.sp.write('N'+str(ncard))
        self.sp.write('r'); response = self.sp.read(4)
        first_word = int(cleanUpString(response), 16)
        self.sp.write('t'); response = self.sp.read(1)
        time_base = int(response)
        if time_base == 0:
            # this is an assumption built into the hardware
            time_base = 1
            time_scale = 4
        else:
            # time_base is already correct
            time_scale = 1
        self.sp.flushInput()
        return {'sampling': sampling, 'time_base': time_base,
                'time_scale': time_scale, 'first_word': first_word}

    def get_timebase_settings(self, unit, print_it=0):
        """
        Returns a dictionary containing the timebase settings.
        """
        self.sp.flushInput()
        self.sp.write('d'+str(unit)); response = self.sp.read(4)
        pretrigger_samples = int(response, 10)
        self.sp.write('b'+str(unit)); response = self.sp.read(1)
        if response.isdigit():
            buffer_size = int(response) * 1024
        else:
            buffer_size = 0
        self.sp.write('p'+str(unit)); response = self.sp.read(2)
        try:
            sample_period = int(response)
        except:
            sample_period = 0
        self.sp.write('m'+str(unit)); response = self.sp.read(1)
        if response == '1':
            multiplier = 100
        else:
            multiplier = 1
        self.sp.write('u'+str(unit)); response = self.sp.read(1)
        trigger_unit = int(response)
        #
        info = {'buffer_size': buffer_size,
                'pretrigger_samples': pretrigger_samples,
                'sample_period': sample_period*multiplier,
                'trigger_unit': trigger_unit}
        if print_it:
            print "Timebase", unit, ":", info
        self.sp.flushInput()
        return info

    def get_timebase_status(self, print_it=0):
        """
        Returns the details for all timebases in a dictionary.
        """
        resultDict = {}
        for time_base in [1,2,3]:
            resultDict[time_base] = self.get_timebase_settings(time_base)
        return resultDict

    def read_data_buffer(self, ncard, channel):
        """
        Read the data buffer for a particular card and channel.
        """
        self.sp.flushInput()
```

```
        self.sp.write('N'+str(ncard))
        self.sp.write('D'+str(channel))
        expected_nchar = 20 + 16*1024 + 8
        text_data = self.sp.read(expected_nchar)
        self.sp.flushInput()
        received_checksum = text_data[-8:-4]
        reserved_trailer = text_data[-4:]
        if len(text_data) != expected_nchar or reserved_trailer != 'zzzz':
            print "Response contains", len(text_data), \
                  "characters, expected", expected_nchar
            print "checksum:", received_checksum, "trailer:", reserved_trailer
            return None
        else:
            return text_data

    def decode_data_buffer(self, text_data, first_word=0,
                           buffer_size=8192, mux=0, print_it=0):
        """
        Returns a list of signals decoded from the data_buffer bytes.
        Each signal consists of a list of voltage samples.

        text_data   : string of bytes from the USB controller
        first_word  : pointer to the oldest word in the ring buffer
                      Note that the USB controller sends the data through
                      in correct order so a value of first_word=0 is default.
        buffer_size : number of sampled data points
        mux         : number of multiplexed channels
                      0 == no multiplexing
                      1, 2, 3, 4 == this many signals through the multiplexer
        The convention for decoding multiplexed signals is that
        the first subchannel will have a higher starting level
        than the others.
        This will have been manually set on the multiplexer box and
        is not communicated trhough the electronics.
        """
        # Split the text into sections.
        header = text_data[0:20]
        data_string = text_data[20:20+(16*1024)]
        received_checksum = int(text_data[-8:-4], 16)
        print "Card:", header[0], "Channel:", header[1], "header:", header
        #
        # Pull a couple items out of the header
        FS = float(header[17:20])
        ground_input = int(header[12]) # raw slope and coupling
        #
        # First, get all of the sampled data into one array.
        try:
            # For the newer numpy
            voltage = zeros((buffer_size,), float)
        except:
            # For the older Numeric
            voltage = zeros((buffer_size,), Float)
        chksum = 0
        for i in range(buffer_size):
            word_pointer = (first_word + i) % 8192
            low_byte = ord(data_string[word_pointer*2])
            high_byte = ord(data_string[word_pointer*2 + 1])
            int_value = (high_byte-0x30)*64 + (low_byte-0x30)
            chksum = rotate_right(chksum)
            chksum += int_value
            chksum &= 0xffff
            sample_volts = (float(int_value)/2048.0 - 1.0) * FS
            voltage[i] = sample_volts
            if i < 3 and print_it > 0:
                print "i=", i, "bytes=", low_byte, high_byte, \
                      "volts=", sample_volts
        if chksum == received_checksum:
            print "Check sums matched. OK."
```

```
        else:
            print "Check sums did not match:", received_checksum, chksum
        if mux <= 1:
            # One signal only, even if it goes through the multiplexer.
            v_demux_sorted = [voltage,]
        else:
            # For two or more multiplexed signals,
            # separate and then decide which is first.
            # We will end up with a list of signals.
            v_demux = []
            for m in range(mux):
                v_demux.append(voltage[m::mux])
            # It is expected that the first subchannel has the
            # biggest initial voltage level.
            biggest = 0  # Starting guess only.
            biggest_sumv = sum(v_demux[0][5:25])
            for m in range(1,mux):
                sumv = sum(v_demux[m][5:25])
                if sumv > biggest_sumv:
                    biggest = m
                    biggest_sumv = sumv
            # Put in order, starting with the biggest signal.
            v_demux_sorted = []
            for m in range(mux):
                v_demux_sorted.append(v_demux[(biggest+m) % mux])
        return (v_demux_sorted, ground_input, FS)


#------------------------------------------------------------------------

if __name__ == '__main__':
    print "Test dbox_services_usb.py..."
    def delay_seconds(n=1):
        import time
        t0 = time.time()
        while time.time() < t0+n: pass
        return

    dbox = BCDDataBox()
    if dbox.sp == None:
        print "Did not find a databox; exiting."
        sys.exit()

    for unit in [1,2,3]:
        print "trigger unit", unit, ":", \
              dbox.get_trigger_unit_settings(unit)

    for unit in [1,2,3]:
        print "timebase unit", unit, ":", \
              dbox.get_timebase_settings(unit)

    card_list = []
    for n in range(1,8):
        if dbox.card_is_present(n):
            print "Card", n, "is present"
            card_list.append(n)
        else:
            print "Card", n, "is missing"

    for card in card_list:
        print "Card", card, "status:", dbox.get_card_status(card)

    print "still_sampling:", dbox.still_sampling(card_list)
    print "Synchronize cards, arm and then trigger box"
    dbox.reset_cards_with_list_and_arm([1,7])
    print "still_sampling:", dbox.still_sampling(card_list)
    delay_seconds(3)
    print "send trigger signal"
```

```
    dbox.trigger()
    delay_seconds(3)
    print "still_sampling:", dbox.still_sampling(card_list)

    text_data = dbox.read_data_buffer(ncard=1, channel=1)
    v_sorted, gnd_input, FS = dbox.decode_data_buffer(text_data, print_it=1)
    print "gnd_input:", gnd_input, "FS:", FS
    print "v_sorted:"

    dbox.finalize()
```

B.3 dbox_data_usb.py

```
## \file dbox_data.py
## \brief Data definitions and services for the BCD Databox Viewer program
## \author Peter Jacobs
## \version 1.0, 22-Oct-2004, separated out of dbox_view.py
##          1.01, 4-Apr-2005, Rainer's more complete CSV output is included
##          1.02,10-Jan-2007, Small changes to accommodate the new USB module
##

import sys
import Tkinter
import tkFileDialog
import tkSimpleDialog
import Pmw
import mx.DateTime
try:
    from Numeric import *
except:
    try:
        from numpy import *
    except:
        print "Failed to import either numpy or Numeric"
from dbox_services_usb import *
from my_progress import myProgressDialog

#----------------------------------------------------------------------------

class SignalInfo:
    "Information for a particular data signal."
    def __init__(self, name="noname",
                 card_id=0,
                 channel_id=0,
                 subchannel_id=0,
                 external_gain=1.0,
                 sensitivity=1.0,
                 units="volts",
                 position=0.0,
                 serial_number = "unknown",
                 transducer_type = "unknown",
                 print_it=0):
        self.name            = name
        self.card_id         = card_id
        self.channel_id      = channel_id
        self.subchannel_id   = subchannel_id
        self.external_gain   = external_gain
        self.sensitivity     = sensitivity
        self.units           = units
        self.position        = position
        self.serial_number   = serial_number
        self.transducer_type = transducer_type
        self.raw_data        = None
        self.scaled_data     = None
        self.FS              = 0.0
        self.offset          = 0.0
        self.dt              = 0.0
        self.t0              = 0.0
```

```
        if print_it:
            print "Signal name=%s card_id=%d channel_id=%d subchannel=%d" % \
                  (self.name, self.card_id, self.channel_id, self.subchannel_id)
            print "    external_gain=%f sensitivity=%e units=%s" % \
                  (self.external_gain, self.sensitivity, self.units)
            print "    position=%f serial_number=%s transducer_type=%s" % \
                  (self.position, self.serial_number, self.transducer_type)
        return

class CardInfo:
    "Information for a particular A/D converter card."
    def __init__(self, card_id):
        self.id_number = card_id
        self.used = 0
        self.channels_used = {1:0, 2:0, 3:0}
        # Each A/D channel may have a number of subchannels
        # as specified in the following dictionary.
        self.subchannels = {1:0, 2:0, 3:0}
        self.dt_sample = 0.0
        return

class DataBoxInfo:
    "Management information for the whole databox"
    def __init__(self, dbox_service):
        self.shotId = "shot"
        self.dataDir = "data"
        self.backupDataDir = "data-backup"
        fname = "run_description.template"
        if os.path.exists(fname):
            f = open(fname, "r")
            self.runDescription = f.read()
            f.close()
        else:
            self.runDescription = \
"""Project........
Run number.....
Date..........%s
Blame..........
""" % mx.DateTime.now()
        fname = "signal.config"
        if os.path.exists(fname):
            f = open(fname, "r")
            self.signalConfigText = f.read()
            f.close()
        else:
            self.signalConfigText = ""
        self.signalDict = {}
        self.cardDict = {}
        self.lock = 0
        try:
            if dbox_service.databox_is_present():
                self.databoxIsPresent = 1
            else:
                self.databoxIsPresent = 0
        except Exception:
            print "Something unusual is wrong with the Databox or its device driver."
            junk = raw_input("Press RETURN to continue...")
            # Continue on to allow non-databox related activities such as
            # data-file management, etc
            self.databoxIsPresent = 0
        scan_signal_config(self, dbox_service)
        return

    # We are going to use the lock to allow the check_status functions
    # to be automatically scheduled and, if the system is busy doing something
    # else, we will be able see that and skip on the check_status.
    def wait_to_acquire_lock(self):
        "Block until the lock is available."
```

```
        while self.lock > 0:
            pass
        self.lock += 1
        return self.lock

    def release_lock(self):
        "Give back the lock."
        if self.lock > 0:
            self.lock -= 1
        return self.lock

    def lock_is_free(self):
        return self.lock == 0

#-----------------------------------------------------------------------

def do_save_all_data(dbox_info):
    """
    Write the data to a gzip-files in a subdirectory named after the shotId.

    This also writes the current run description text.
    Should check for the existence of old data files.
    """
    import gzip
    for dataDir in [dbox_info.dataDir, dbox_info.backupDataDir]:
        shotId = dbox_info.shotId
        shotDir = os.path.join(dataDir, shotId)
        if not os.path.isdir(dataDir):
            os.mkdir(dataDir)
        if not os.path.exists(shotDir):
            os.mkdir(shotDir)

        # Run description text is added to the same directory as
        # the data files.
        file_for_runDesc = shotId + ".txt"
        frd = open(os.path.join(shotDir, file_for_runDesc), "wt")
        frd.write(dbox_info.runDescription)
        frd.close()

        # Signal configuration text is added to the same directory as
        # the data files.
        file_for_signalConfig = shotId + ".config"
        fsc = open(os.path.join(shotDir, file_for_signalConfig), "wt")
        fsc.write(dbox_info.signalConfigText)
        fsc.close()

        # Write the data files, one signal at a time.
        # For each data file, add an entry to the list file.
        file_for_list = shotId + "A.LST.gz"
        flist = gzip.open(os.path.join(shotDir, file_for_list), "wb")
        pd = myProgressDialog(maxValue=len(dbox_info.signalDict.keys()),
                              title="dbox_view",
                              message="Writing data to files.")
        for signalId in dbox_info.signalDict.keys():
            signal = dbox_info.signalDict[signalId]
            extn = str(signalId[0]) + str(signalId[1]) + str(signalId[2])
            flist.write("%s %s\n" % (extn, signal.name) )
            fileName = shotId + "A." + extn + ".gz"
            print "Writing data file for signal", signalId, "fileName", fileName
            fdata = gzip.open(os.path.join(shotDir, fileName), "wb")
            write_scaled_data_to_file(dbox_info, signal, fdata, extn)
            fdata.close
            pd.progress += 1
        flist.close()
        pd.close()
        print "Finished writing data and list files to directory=", dataDir
    #
    return
```

```
def do_export_csv_data(dbox_info):
    """
    Write the data to a CSV file in a subdirectory named after the shotId.

    Rainer Kirchhartz' arrangement for writing everything to the CSV file.
    """
    print "Begin exporting data to CSV file."
    dataDir = dbox_info.dataDir
    shotId = dbox_info.shotId
    shotDir = os.path.join(dataDir, shotId)
    file_for_CSV = shotId + ".csv"
    f = open(os.path.join(shotDir, file_for_CSV), "wt")
    signalIdList = dbox_info.signalDict.keys()
    signalIdList.sort()
    f.write("dateTime,")
    for signalId in signalIdList:
        f.write("%s," % mx.DateTime.now())
    f.write("\nshot_id,")
    for signalId in signalIdList:
        f.write("%s," % (dbox_info.shotId,))
    f.write("\nsignal_id,")
    for signalId in signalIdList:
        extn = str(signalId[0]) + str(signalId[1]) + str(signalId[2])
        f.write("%s," % (extn,))
    f.write("\nwithtimecolumn,")
    for signalId in signalIdList:
        f.write("no,")
    f.write("\ndatapoints,")
    max_nsample = 0
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        nsample = len(signal.scaled_data)
        f.write("%d," % (nsample,) )
        max_nsample = max(max_nsample, nsample)
    f.write("\ndataType,")
    for signalId in signalIdList:
        f.write("scaled," )
    f.write("\ndataunits,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.units,) )
    f.write("\ntimestart,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%f," % (signal.t0,) )
    f.write("\ntimeAverageWindow,")
    for signalId in signalIdList:
        f.write("0.0,")
    f.write("\ntimeInterval,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%f," % (signal.dt,) )
    f.write("\ntimeUnits,")
    for signalId in signalIdList:
        f.write("microseconds,")
    f.write("\ntransducerSensitivity,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%f," % (signal.sensitivity,) )
    f.write("\ntransducerSensitivityUnits,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.units,) )
    f.write("\ntransducer_name,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.name,) )
```

```
    f.write("\nposition,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%f," % (signal.position,) )
    f.write("\ntransducerSerialNumber,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.serial_number,) )
    f.write("\ntransducerType,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.transducer_type,) )
    f.write("\ngain,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.external_gain,) )
    f.write("\nqfluxgain,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("1.0,")
    f.write("\nfullScaleVolts,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.FS,) )
    f.write("\noffsetVolts,")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.offset,) )
    f.write("\n")
    f.write("\n")
    for i in range(max_nsample):
        f.write("%d" % i)
        f.write(",")
        for signalId in signalIdList:
            signal = dbox_info.signalDict[signalId]
            if i < len(signal.scaled_data):
                f.write("%f," % (signal.scaled_data[i],) )
            else:
                f.write(",")
        f.write("\n")
    f.close()
    print "Finished exporting data to CSV file."
    #
    return

def do_export_csv_data_old_version(dbox_info):
    """
    Write the data to a CSV file in a subdirectory named after the shotId.
    """
    print "Begin exporting data to CSV file."
    dataDir = dbox_info.dataDir
    shotId = dbox_info.shotId
    shotDir = os.path.join(dataDir, shotId)
    file_for_CSV = shotId + ".csv"
    f = open(os.path.join(shotDir, file_for_CSV), "wt")
    signalIdList = dbox_info.signalDict.keys()
    signalIdList.sort()
    for signalId in signalIdList:
        extn = str(signalId[0]) + str(signalId[1]) + str(signalId[2])
        f.write("%s," % (extn,))
    f.write("signal_id\n")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.name,) )
    f.write("signal_name\n")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%s," % (signal.units,) )
```

```
    f.write("units\n")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%f," % (signal.position,) )
    f.write("position\n")
    max_nsample = 0
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        nsample = len(signal.scaled_data)
        f.write("%d," % (nsample,) )
        max_nsample = max(max_nsample, nsample)
    f.write("nsample\n")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%d," % (signal.t0,) )
    f.write("t0\n")
    for signalId in signalIdList:
        signal = dbox_info.signalDict[signalId]
        f.write("%d," % (signal.dt,) )
    f.write("dt\n")
    for i in range(max_nsample):
        for signalId in signalIdList:
            signal = dbox_info.signalDict[signalId]
            if i < len(signal.scaled_data):
                f.write("%f," % (signal.scaled_data[i],) )
            else:
                f.write(",")
        f.write("%d\n" % i)
    f.close()
    print "Finished exporting data to CSV file."
    #
    return

#------------------------------------------------------------------------

def scale_signal_data(signal):
    "Scale the raw data using the sensitivity and the initial level as zero."
    N = 20
    start_level = sum(signal.raw_data[5:5+N])/float(N)
    signal.offset = start_level
    signal.scaled_data = (signal.raw_data - start_level) / signal.sensitivity \
                         / signal.external_gain
    return

def reconstruct_raw_voltages(signal):
    "Presumably we have loaded (old) scaled data from disc."
    signal.raw_data = signal.scaled_data * signal.sensitivity * signal.external_gain \
                      + signal.offset
    return

#------------------------------------------------------------------------

def write_scaled_data_to_file(dbox_info, signal, f, extn):
    "Send the formatted data to already-opened file f."
    f.write("# dataSource dbox_view_usb v1.2 2007\n")
    f.write("# dateTime %s\n" % mx.DateTime.now())
    f.write("# shotName %s\n" % dbox_info.shotId)
    f.write("# channelId %s\n" % extn)
    f.write("# withTimeColumn no\n")
    try:
        nsample = len(signal.scaled_data)
    except:
        nsample = 0
        print "WARNING: channel", extn, "seems to have no data."
    f.write("# dataPoints %d\n" % nsample)
    f.write("# dataType scaled\n")
    f.write("# dataUnits %s\n" % signal.units)
    f.write("# timeStart %e\n" % signal.t0)
```

```
    f.write("# timeAverageWindow 0.0\n")
    f.write("# timeInterval %e\n" % signal.dt)
    f.write("# timeUnits microseconds\n")
    f.write("# transducerSensitivity %e\n" % signal.sensitivity)
    f.write("# transducerSensitivityUnits %s\n" % signal.units)
    f.write("# transducerName %s\n" % signal.name)
    f.write("# transducerLocation %e\n" % signal.position)
    f.write("# transducerSerialNumber %s\n" % signal.serial_number)
    f.write("# transducerType %s\n" % signal.transducer_type)
    f.write("# gain %e\n" % signal.external_gain)
    f.write("# qfluxgain 1.0\n")
    f.write("# fullScaleVolts %e\n" % signal.FS)
    f.write("# offsetVolts %e\n" % signal.offset)
    f.write("\n")
    for i in range(nsample):
        f.write("%e\n" % signal.scaled_data[i])
    return

def split_metadata(line):
    line = line[1:]  # throw away the sharp character
    line = line.strip()
    words = line.split()
    name = words[0]
    try:
        stringvalue = words[1]
    except:
        stringvalue = "unknown"
    return name, stringvalue

def read_data_from_file(dbox_info, signal, f, extn):
    "Get the formatted data from the already-opened file f."
    # Default data, in case the file does not tell us all...
    withTimeColumn = 0
    nsample = 0
    signal.units = "Volts"
    signal.t0 = 0.0
    signal.dt = 1.0
    signal.name = "None"
    signal.position = 0.0
    signal.serial_number = "1234"
    signal.transducer_type = "unknown"
    signal.external_gain = 1.0
    signal.FS = 5.0
    signal.offset = 0.0
    # Process the header
    line = f.readline().strip()
    while line[0] == "#":
        # process a line of metadata
        name, stringvalue = split_metadata(line)
        print "metadata: name=", name, " value=", stringvalue
        # if name == "shotName": dbox_info.shotId = stringvalue
        if name == "withTimeColumn": withTimeColumn = stringvalue
        if name == "dataPoints": nsample = int(stringvalue)
        if name == "dataUnits": signal.units = stringvalue
        if name == "timeStart": signal.t0 = float(stringvalue)
        if name == "timeInterval": signal.dt = float(stringvalue)
        if name == "transducerSensitivity": signal.sensitivity = float(stringvalue)
        if name == "transducerSensitivityUnits": signal.units = stringvalue
        if name == "transducerName": signal.name = stringvalue
        if name == "transducerLocation": signal.position= float(stringvalue)
        if name == "transducerSerialNumber": signal.serial_number = stringvalue
        if name == "transducerType": signal.transducer_type = stringvalue
        if name == "gain": signal.external_gain = float(stringvalue)
        if name == "fullScaleVolts": signal.FS = float(stringvalue)
        if name == "offsetVolts": signal.offset = float(stringvalue)
        # Read the following line for the next iteration, if any.
        line = f.readline().strip()
        if len(line) == 0: break
```

```
    # We should have come across the blank line separating the
    # file header from the file data.
    print "Read sampled data"
    try:
        # For the older Numeric
        signal.scaled_data = zeros((nsample,), Float)
    except:
        # For the newer numpy
        signal.scaled_data = zeros((nsample,), float)
    for i in range(nsample):
        line = f.readline().strip()
        if withTimeColumn == "no":
            # Only one number on the line.
            samplestring = line
        else:
            # Ignore the timestamp.
            samplestring = line.split()[1]
        try:
            signal.scaled_data[i] = float(samplestring)
        except:
            break
    print "Number of samples read", len(signal.scaled_data)
    return

#-----------------------------------------------------------------------

def scan_signal_config(dbox_info, dbox_service):
    """
    Scans the information for each configured signal from text form.

    There should be one line for each signal to be configured.
    Lines starting with a hash (or sharp) character are comments
    (and are ignored).
    """
    # Clear out old config data
    dbox_info.cardIdList = []
    dbox_info.cardDict = {}
    dbox_info.signalDict = {}

    print "---------------------------------------"
    print "Scan Signal Configuration..."
    for line in dbox_info.signalConfigText.split("\n"):
        line = line.strip()
        if len(line) == 0: continue
        if line[0] == "#": continue
        # process the noncomment line
        # The definition of what is expected on a line is given by the
        # following lines. Always check here for the latest definition.
        elements = line.split()
        try:
            signal = SignalInfo( name            = elements[0],
                                 card_id         = int(elements[1]),
                                 channel_id      = int(elements[2]),
                                 subchannel_id   = int(elements[3]),
                                 external_gain   = float(elements[4]),
                                 sensitivity     = float(elements[5]),
                                 units           = elements[6],
                                 position        = float(elements[7]),
                                 serial_number   = elements[8],
                                 transducer_type = elements[9],
                                 print_it        = 1)
        except:
            print "Something is wrong with this signal config line:"
            print line
            print "Will ignore this signal definition and continue with the next."
            continue
        # Each signal is identified by the tuple.
        signalId = (signal.card_id, signal.channel_id, signal.subchannel_id)
```

```
        # Check that we don't configure the same combination of
        # card, channel and subchannel twice.
        if dbox_info.signalDict.has_key(signalId):
            print "Configuration error: already done", signalId
            print "Will ignore this redundant definition and continue with the next."
            continue
        else:
            dbox_info.signalDict[signalId] = signal
        if not dbox_info.cardDict.has_key(signal.card_id):
            # New card, add it to the dictionary of cards.
            card = CardInfo(signal.card_id)
            card.used = 1
            dbox_info.cardDict[signal.card_id] = card
        else:
            # Card already exists, get a reference to it.
            card = dbox_info.cardDict[signal.card_id]
        card.channels_used[signal.channel_id] = 1
        if signal.subchannel_id > 0:
            # This signal is part of a multiplexed set.
            card.subchannels[signal.channel_id] += 1
        # end of processing noncomment line
    # end of loop body for each line
    print "Configured cards and channels:"
    for id in dbox_info.cardDict.keys():
        card = dbox_info.cardDict[id]
        print "Card:", card.id_number, \
              "Channels used:", card.channels_used, \
              "SubChannel counts:", card.subchannels
        if not dbox_service.card_is_present(id):
            print "WARNING: this card is not present in the databox."
    print "-----------------------------------------------"
    return

def rescan_signal_config_for_new_scales(dbox_info):
    """
    Keep the old signals but update the signal specifications such as
    transducer sensitivity, etc.
    """
    print "---------------------------------------"
    print "Rescan Signal Configuration for new sensitivities, etc."
    for line in dbox_info.signalConfigText.split("\n"):
        line = line.strip()
        if len(line) == 0: continue
        if line[0] == "#": continue
        # process the noncomment line
        elements = line.split()
        # Each signal is identified by the tuple.
        card_id       = int(elements[1])
        channel_id    = int(elements[2])
        subchannel_id = int(elements[3])
        signalId = (card_id, channel_id, subchannel_id)
        signal = dbox_info.signalDict[signalId]
        signal.external_gain = float(elements[4])
        signal.sensitivity   = float(elements[5])
        signal.units         = elements[6]
        signal.position      = float(elements[7])
        signal.serial_number = elements[8]
        signal.transducer_type = elements[9]
    print "---------------------------------------"
    return

#---------------------------------------------------------------------------------------------
```