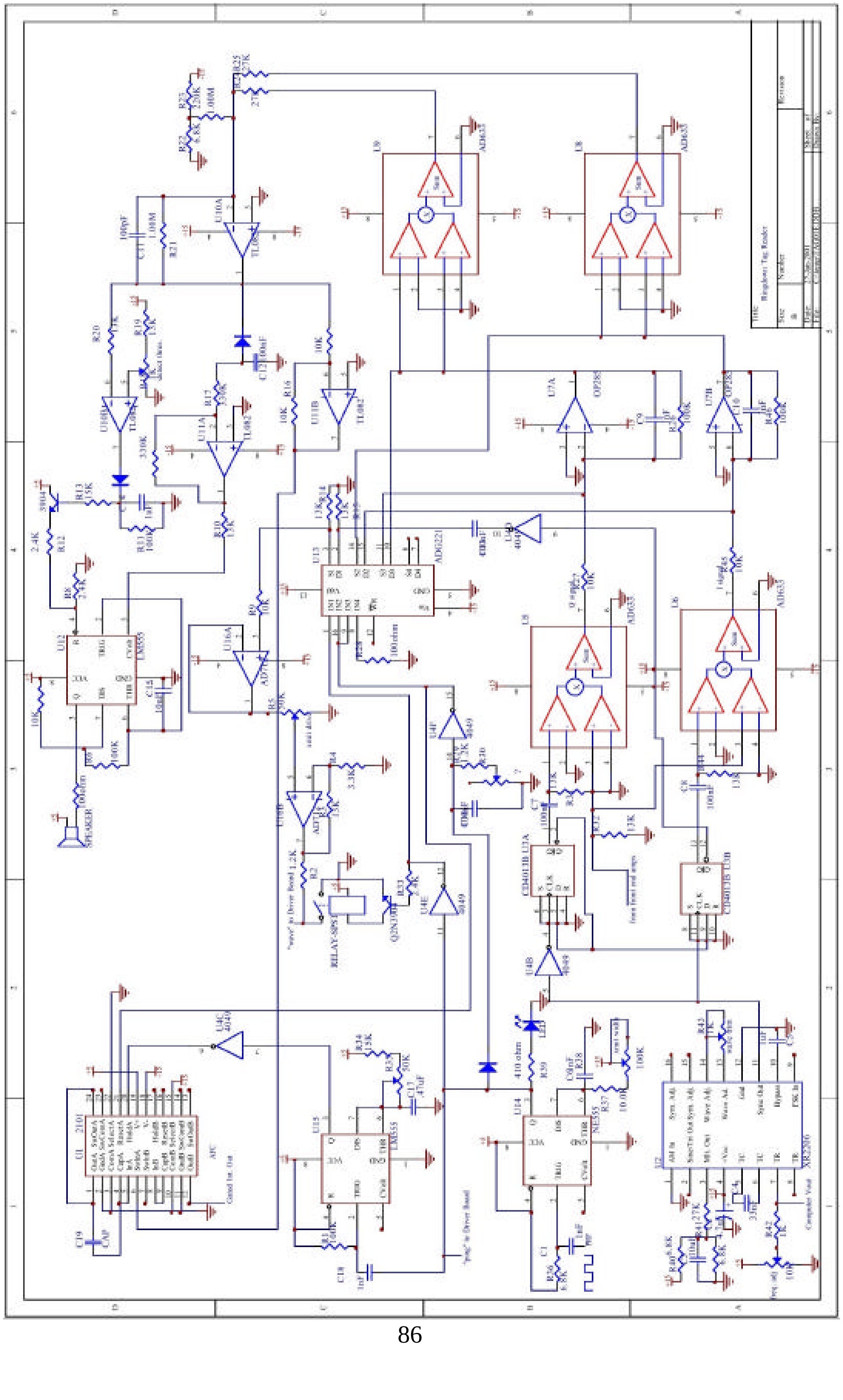
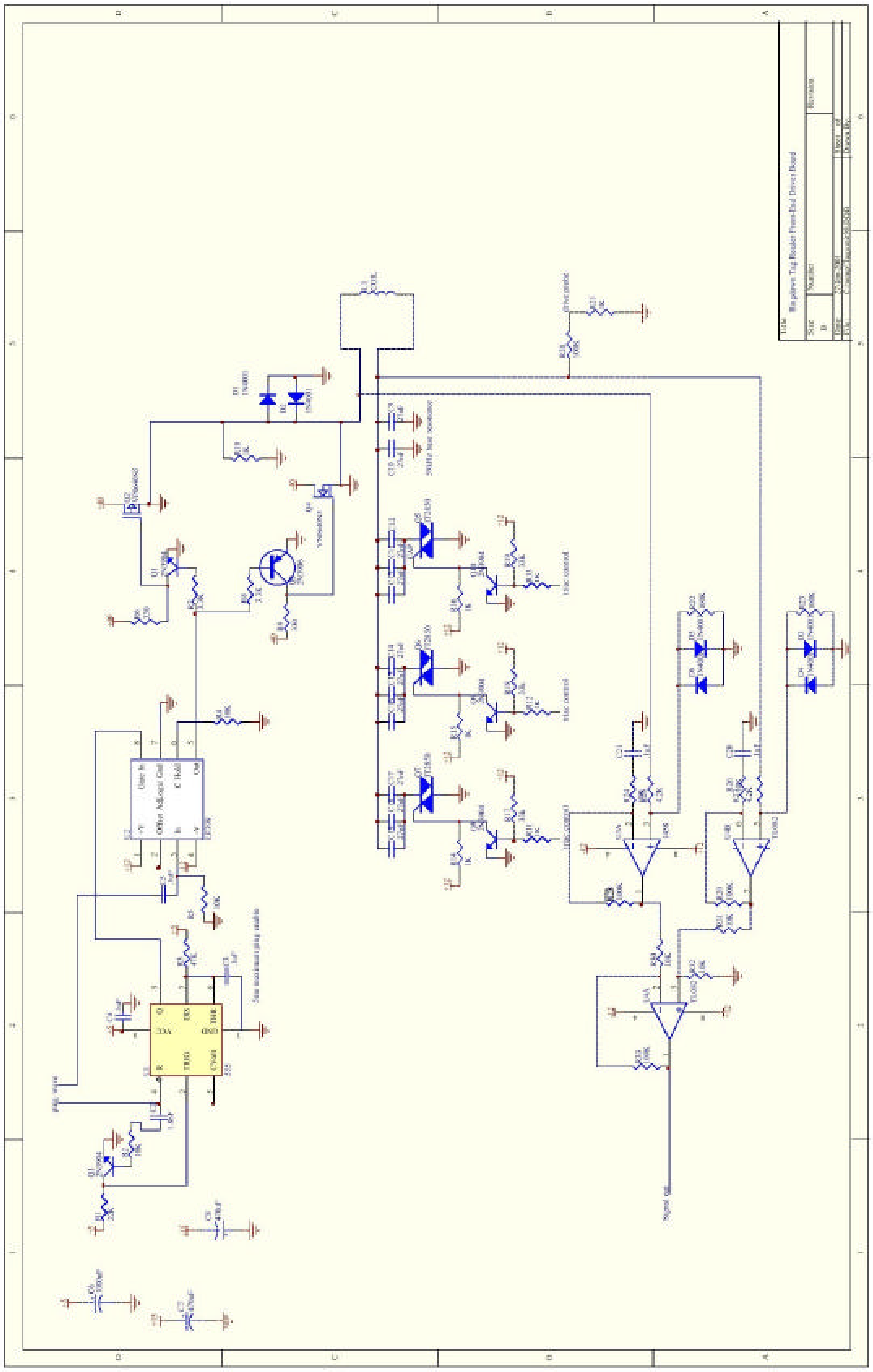
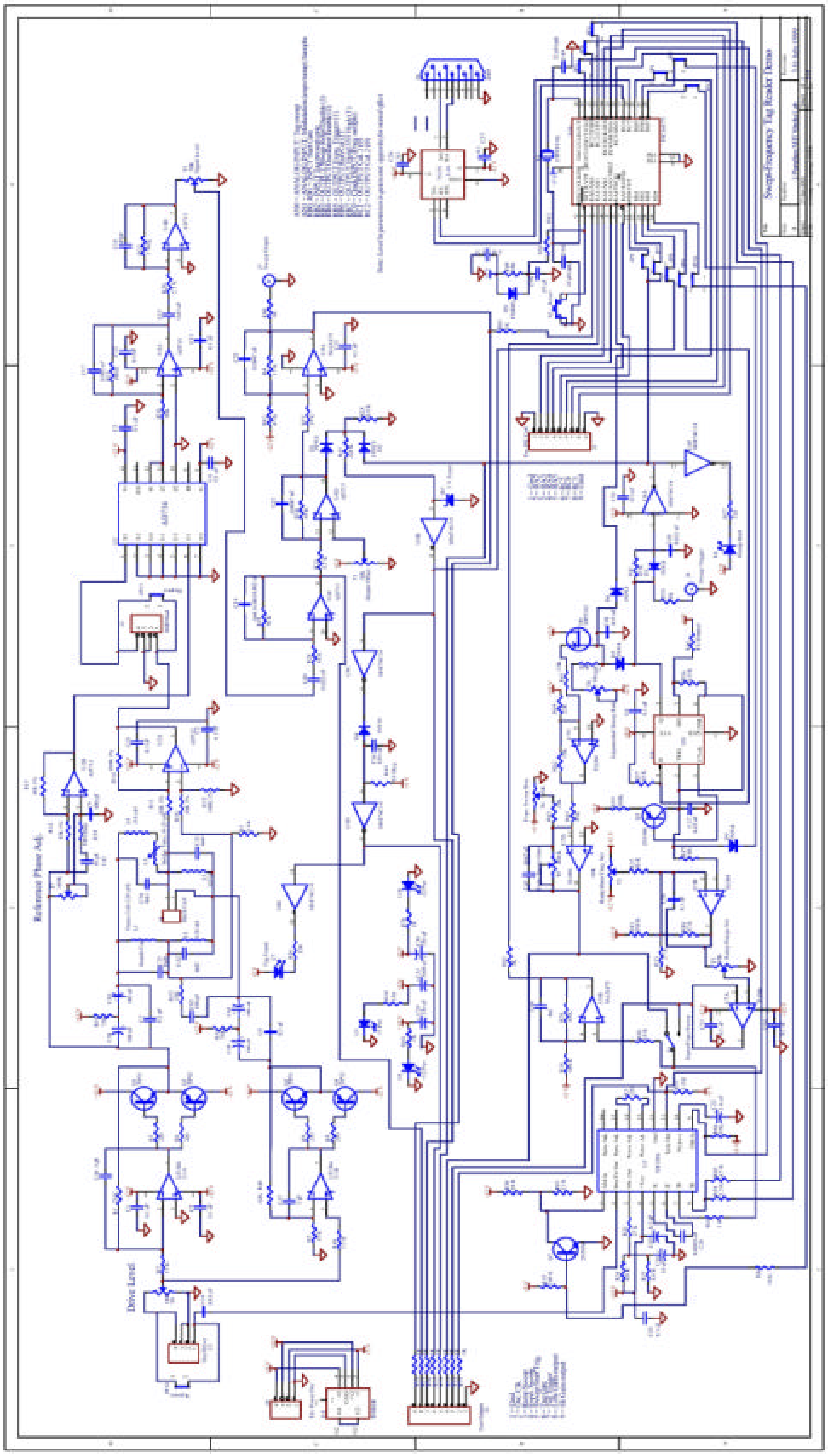
Appendix A

This appendix presents the schematics for the ringdown tag reader, the ringdown tag reader front end driver, and the swept-frequency tag reader.







Swept-Frequency Tag Reader Demo
Reference Phase Adj.
Drive Level

Appendix B

Here we present the information necessary to implement the frequency drift compensation algorithms described in section 3.5.

On each normal sweep, the tag reader samples the output voltage of the tag system several hundred times, and for each peak that exceeds the baseline threshold, it outputs relevant information. For one sweep, the serial transmission provides the following information:

1. The start time (where "time" is the counter of samples taken so far this sweep) of each peak,
2. The stop time of each peak,
3. The sum of all samples under the peak (essentially an integral),
4. The number of samples taken in the course of the entire sweep.

By using frequency information and the number of samples to adjust the measured start time and stop time, we can compensate for frequency drift.

On each frequency calibration sweep, the tag reader outputs samples of the frequency at periodic intervals during the sweep. In a typical sweep, ten samples are returned. When more samples are returned we can ignore the samples beyond the tenth; although these are part of the sweep, tags detected beyond the tenth sample may not be picked up under all common temperature conditions (one of the timing issues discovered while taking heat measurements for section 3.5).

First, let us derive the compensation algorithm for the linear sweep. Let t_{start} be the start time for a tag, t_{stop} be the stop time of the tag, and n be the number of samples taken during an initial calibration sweep for that tag. Further, let us take frequency samples so that f_1 and f_{10} are the first and tenth frequency samples in that initial calibration sweep. Then, our host computer's program can store a value for the start and stop frequencies of the tag that has been converted from the arbitrary counter timing of the PIC code into something more closely resembling the actual frequencies of the tag.

This can be done by scaling each time *t* such that $f_{real} = \frac{t}{n}(f_1 - f_{10}) + f_{10}$. After this, our host computer can perform its computations by identifying tags based on their true frequencies. Then, as time passes, temperatures change, and the frequency range drifts, we can take new frequency samples periodically. Based on the new frequency measurements, each subsequent incoming tag start and end time can be scaled also by the equation just given, and tag identification can proceed based on the new scaled frequencies, using the same lookup table as the one generated on startup.

The compensation algorithm for the exponential sweep works exactly the same way, except that three frequencies are necessary to scale the tag time into an actual frequency. By using f_1, f_5, and f_9, the first, fifth, and ninth frequency samples, we can scale a time given by the PIC into a real frequency once we can derive the exponential function which would produce f_1, f_5, and f_9. We use these particular values because they are spaced equidistantly in time, which simplifies the math. By scaling the time axis so f_1 occurs at time 0, f_5 occurs at time 1, and f_9 occurs at time 2, we find we have to solve for A, B, and k in a system of three equations

$$Ae^{-0k} + B = f_1$$

$$Ae^{-k} + B = f_5$$

$$Ae^{-2k} + B = f_9$$

Taking logarithms of the second and third equation, it becomes possible to scale both equations to be equal to k. Removing k and then substituting B using the first equation, we obtain

$$A = \frac{(f_5 - f_1)^2}{2(f_9 - f_5)}$$

Thanks to the first equation, we thus know that

$$B = f_1 - A$$

and finally we have

$$k = -\ln \frac{f_5 - B}{A}$$

Given the values of A, B, and k, we can derive a number analogous to the actual frequency for each reported tag start and stop time t using the relation

$$f_{real} = Ae^{-20kt/9n} + B$$

As with the linear case, by recording the values of f_{real} for each t reported by the tag reader, we can then use these values to distinguish between tags by their frequency in the host computer, and when frequency measurements appear in periodic calibration updates, it is possible to adjust the coefficients accordingly using the above relations, which then continue to allow subsequent values to be scaled.

With a system like this implemented, it should then be possible to run the swept-frequency tag reader for extended periods of time without the need to recalibrate for the frequency drift of the reader. Frequency drift in the tags themselves is much harder to compensate, but apart from extreme heat and physical distortion, the tags tend to be more stable than the reader, particularly because the tags are in open air at room temperature, while the reader board is often near other equipment (e.g. video projectors) which alters its ambient temperature more often.

Appendix C

In this appendix we present the code used in the various systems described in this thesis. The final PIC used with the PIC code was the Microchip PIC16F873. C code was compiled using CCS PIC C (see http://www.ccsinfo.com) and written using MPLAB (see http://www.microchip.com) using a PICSTART Plus chip programmer (also available from Microchip).

Windows C++ code was written and compiled in Microsoft Visual Studio, with the addition of Rogus McBogus, a MIDI library developed by the Brain Opera group at the Media Lab, and with GLUT, the extension library for OpenGL (information available at http://reality.sgi.com/opengl/glut3/glut3.html). The code used for the Musical Trinkets is very long and will not be included here. Links and information are current as of January 2001; contact the author for further information.

C.1. Swept-Frequency Master Board PIC

The first program is the one used to control the PIC on the master tag reader board in the six-coil tag system, which can be used as is to control a single tag reader as well.

```
#include "tag10pic.h"

#define PIN_TRIGGER PIN_B1
#define PIN_CAL 1
#define PIN_INPUT 0
#define PIN_EXPORESET PIN_B0
#define PIN_TAGPRESENT PIN_B2
#define PIN_RESETSWEEP PIN_B3
#define PIN_OSCILLATE PIN_B4
#define PIN_EXPOTRIG PIN_B5
#define PIN_RUNCAL PIN_B6
#define PIN_HI555 PIN_B7
#define PIN_GATE PIN_C1
#define PIN_MUX0 PIN_C4
```

```
#define PIN_MUX1 PIN_C5

#fuses HS,NOWDT,NOPROTECT,PUT,NOBROWNOUT
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
#use fast_io(B)
#use fast_io(C)

typedef union {
  unsigned long two;
  unsigned int ones[2];
} WORD;

unsigned int i;
WORD tmp;
unsigned int overflows=0;
unsigned long threshold=0;
unsigned int counter=0;
unsigned int last, start;
char axis=0; // which pair of boards are we running now?  control the mux
int running=1;
char sweeping=1;
WORD sum;
unsigned int bufstart=0, bufend=0, buffer[80];

void addbuf(unsigned int val)
{
  if (bufend==bufstart-1 || (bufend==79 && bufstart==0)) return;
  buffer[bufend++]=val;
  if (bufend>79) bufend=0;
}

unsigned int getbuf()
{
   unsigned int val;
   if (bufend==bufstart) return 255;
   val=buffer[bufstart++];
   if (bufstart>79) bufstart=0;
   return val;
}

main()
{
#asm
bsf 0x9f, 7
#endasm
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
   setup_counters(RTCC_INTERNAL,RTCC_DIV_2);
   setup_port_a(ALL_ANALOG);
   setup_adc(ADC_CLOCK_INTERNAL);
   set_tris_b(0b00000111);
   set_tris_c(0b10000000);
   printf("PIC starting up...\n");
   output_high(PIN_RESETSWEEP);
   output_high(PIN_OSCILLATE);
   output_high(PIN_EXPOTRIG);
   output_high(PIN_RUNCAL);
   output_high(PIN_HI555);
   output_low(PIN_MUX0);
   output_low(PIN_MUX1);
   while (!input(PIN_TRIGGER));
   while (input(PIN_TRIGGER));
   for (i=0; i<127; i++)
   {
      set_adc_channel(PIN_INPUT);
      delay_us(45);
      while (!input(PIN_TRIGGER))
      {
         tmp.two=read_adc();
         if (tmp.two>threshold)
            threshold=tmp.two;
      }
      printf("%ld ", threshold);

      while (input(PIN_TRIGGER));
   }


  setup_counters(RTCC_INTERNAL, RTCC_DIV_4);
  enable_interrupts(RTCC_ZERO);
  enable_interrupts(GLOBAL);

   while(1)
   {
      set_adc_channel(PIN_INPUT);
      while (!input(PIN_TRIGGER));
      disable_interrupts(GLOBAL);
      addbuf(overflows);
      addbuf(counter);
      addbuf(255);
      addbuf(255-axis);
      counter=0;
      overflows=0;

      axis++;
      if (axis>2) axis=0;
      if (axis==1) output_high(PIN_MUX0); else output_low(PIN_MUX0);
      if (axis==2) output_high(PIN_MUX1); else output_low(PIN_MUX1);
while (input(PIN_TRIGGER))
      {
         if (bufstart!=bufend)
         {
            printf("%c", getbuf());
         }
enable_interrupts(GLOBAL);

   }
}

#INT_RTCC
void loop()
{
         int start_o;
         set_rtcc(100);


         if (counter==255) {overflows++; counter=0;}
         else counter++;

         tmp.two=read_adc();
         if (tmp.two>threshold)
         {
             if (last==0)
             {
                start=counter;
//                start_o=overflows;
             }
             last=1;
             sum.two+=(tmp.two);
         }
         else
         {
            if (last==1)
            {
//                addbuf(start_o);
               addbuf(start);
//                addbuf(overflows);
               addbuf(counter);
               addbuf(sum.ones[1]);
               addbuf(sum.ones[0]);
               last=0;
               sum.two=0;
            }
         }

   if ((!(counter&0b00000011)) && (bufstart!=bufend))              // every 4 counts
   {
      printf("%c", getbuf());
   }

}
```

C.2. Swept-Frequency Frequency-Drift Collector

This next program is similar to the first but its sole purpose is to return frequency and voltage samples from the voltage-controller oscillator. Added to the code in C.1 and combined with the algorithms for the host computer given in Appendix B, automatic frequency drift compensation should be easily implementable.

```
#include "tag10cal.h"

#define PIN_TRIGGER PIN_B1
#define PIN_CAL 1
#define PIN_INPUT 0
#define PIN_EXPORESET PIN_B0
#define PIN_TAGPRESENT PIN_B2
#define PIN_RESETSWEEP PIN_B3
#define PIN_OSCILLATE PIN_B4
#define PIN_EXPOTRIG PIN_B5
#define PIN_RUNCAL PIN_B6
#define PIN_HI555 PIN_B7
#define PIN_GATE PIN_C3
#define PIN_MUX0 PIN_C4
#define PIN_MUX1 PIN_C5

#fuses HS,NOWDT,NOPROTECT,PUT,NOBROWNOUT
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
#use fast_io(B)
#use fast_io(C)

typedef union {
  unsigned long two;
  unsigned int ones[2];
} WORD;

unsigned int i;
WORD tmp;
unsigned int overflows=0;
unsigned long threshold=0;
unsigned int counter=0;
unsigned int last, start;
char axis=0; // which pair of boards are we running now?  control the mux
int running=1;
char sweeping=1;
WORD sum;

void exp_recal()
{
  WORD vtmp1, vtmp2, vstmp1, vstmp2, vstmp3;
  WORD ftmp1, ftmp2, ftmp3;

  // get voltage and frequency 1st time
while (1) {
  while(!input(PIN_TRIGGER))
    ;
  while(input(PIN_TRIGGER))
    ;
  while(!input(PIN_TRIGGER))
    ;
  while(input(PIN_TRIGGER))
    ;
  while(!input(PIN_TRIGGER)) {

    vstmp1.two = read_adc();
    set_timer1(0);
    delay_ms(1);
    ftmp1.two=get_timer1();

    if (input(PIN_TRIGGER)) break; // abort if cycle ended

    printf("%c", vstmp1.ones[1]);
    printf("%c", vstmp1.ones[0]);
    printf("%c", ftmp1.ones[1]);
    printf("%c", ftmp1.ones[0]);
  }
  printf("%c%c", 255, 255);

}
}

main()
{
#asm
bsf 0x9f, 7
#endasm
#use rs232(baud=19200, xmit=PIN_C6, rcv=PIN_C7)
   setup_counters(RTCC_INTERNAL,RTCC_DIV_2);
   setup_port_a(ALL_ANALOG);
   setup_adc(ADC_CLOCK_INTERNAL);
   set_tris_b(0b00000110);
   set_tris_c(0b10000011);
   printf("PIC starting up...\n");
   output_high(PIN_RESETSWEEP);
   output_high(PIN_OSCILLATE);
   output_high(PIN_EXPOTRIG);
   output_high(PIN_RUNCAL);
   output_high(PIN_HI555);
   output_low(PIN_MUX0);
   output_low(PIN_MUX1);
  set_adc_channel(PIN_CAL);

  setup_timer_1(T1_EXTERNAL|T1_DIV_BY_1);
  exp_recal();
}
```

C.3. PIC Header File

This header file is the one included in the programs in both C.1 and C.2.

```
//////// Standard Header file for the PIC16F873 device ////////
#device PIC16F873
#use delay(clock=20000000)
#nolist
////////////////////////////// I/O definitions for INPUT() and OUTPUT_xxx()
#define PIN_A0  40
#define PIN_A1  41
#define PIN_A2  42
#define PIN_A3  43
#define PIN_A4  44
#define PIN_A5  45

#define PIN_B0  48
#define PIN_B1  49
#define PIN_B2  50
#define PIN_B3  51
#define PIN_B4  52
#define PIN_B5  53
#define PIN_B6  54
#define PIN_B7  55

#define PIN_C0  56
#define PIN_C1  57
#define PIN_C2  58
#define PIN_C3  59
#define PIN_C4  60
#define PIN_C5  61
#define PIN_C6  62
#define PIN_C7  63

////////////////////////////// Useful defines
#define FALSE 0
#define TRUE 1

#define BYTE int
#define BOOLEAN short int

#define getc getch
#define getchar getch
#define puts(s) {printf(s); putchar(13); putchar(10);}
#define putc putchar

////////////////////////////// Constants used for RESTART_CAUSE()
#define WDT_FROM_SLEEP  0
#define WDT_TIMEOUT     8
#define MCLR_FROM_SLEEP 16
#define NORMAL_POWER_UP 24
////////////////////////////// Constants used for SETUP_COUNTERS()
#define RTCC_INTERNAL   0
#define RTCC_EXT_L_TO_H 32
```

```
#define RTCC_EXT_H_TO_L 48
#define RTCC_DIV_2      0
#define RTCC_DIV_4      1
#define RTCC_DIV_8      2
#define RTCC_DIV_16     3
#define RTCC_DIV_32     4
#define RTCC_DIV_64     5
#define RTCC_DIV_128    6
#define RTCC_DIV_256    7
#define WDT_18MS        8
#define WDT_36MS        9
#define WDT_72MS       10
#define WDT_144MS      11
#define WDT_288MS      12
#define WDT_576MS      13
#define WDT_1152MS     14
#define WDT_2304MS     15
#define L_TO_H             0x40
#define H_TO_L                0

#define RTCC_ZERO          0x0B20    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_RTCC           0x0B20    // Used for ENABLE/DISABLE INTERRUPTS
#define RB_CHANGE          0x0B08    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_RB             0x0B08    // Used for ENABLE/DISABLE INTERRUPTS
#define EXT_INT            0x0B10    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_EXT            0x0B10    // Used for ENABLE/DISABLE INTERRUPTS

#define GLOBAL             0x0BC0    // Used for ENABLE/DISABLE INTERRUPTS
//////////////////////////////////// Constants used for Timer1 and Timer2
#define T1_DISABLED        0
#define T1_INTERNAL        5
#define T1_EXTERNAL        7
#define T1_EXTERNAL_SYNC   3
#define T1_CLK_OUT         8
#define T1_DIV_BY_1        0
#define T1_DIV_BY_2        0x10
#define T1_DIV_BY_4        0x20
#define T1_DIV_BY_8        0x30
#byte   TIMER_1_LOW=       0x0e
#byte   TIMER_1_HIGH=      0x0f
#define T2_DISABLED        0
#define T2_DIV_BY_1        4
#define T2_DIV_BY_4        5
#define T2_DIV_BY_16       6
#byte   TIMER_2=           0x11

#define INT_TIMER1         0x8C01    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_TIMER2         0x8C02    // Used for ENABLE/DISABLE INTERRUPTS

/////////////////////////////////// Constants used for SETUP_CCP1()
#define CCP_OFF                        0
#define CCP_CAPTURE_FE                 4
#define CCP_CAPTURE_RE                 5
#define CCP_CAPTURE_DIV_4              6
#define CCP_CAPTURE_DIV_16             7
#define CCP_COMPARE_SET_ON_MATCH       8
#define CCP_COMPARE_CLR_ON_MATCH       9
#define CCP_COMPARE_INT                0xA
#define CCP_COMPARE_RESET_TIMER        0xB
#define CCP_PWM                        0xC
#define CCP_PWM_PLUS_1                 0x1c
```

```
#define CCP_PWM_PLUS_2                  0x2c
#define CCP_PWM_PLUS_3                  0x3c
long CCP_1;
#byte   CCP_1    =                      0x15
#byte   CCP_1_LOW=                      0x15
#byte   CCP_1_HIGH=                     0x16

#define INT_CCP1            0x8C04    // Used for ENABLE/DISABLE INTERRUPTS

//////////////////////////////////// Constants used for SETUP_CCP2()
long CCP_2;
#byte   CCP_2    =                      0x1B
#byte   CCP_2_LOW=                      0x1B
#byte   CCP_2_HIGH=                     0x1C

#define INT_CCP2            0x8D01    // Used for ENABLE/DISABLE INTERRUPTS

//////////////////////////////////// Constants used in SETUP_SSP()
#define SPI_MASTER       0x20
#define SPI_SLAVE        0x24
#define SPI_L_TO_H       0
#define SPI_H_TO_L       0x10
#define SPI_CLK_DIV_4    0
#define SPI_CLK_DIV_16   1
#define SPI_CLK_DIV_64   2
#define SPI_CLK_T2       3
#define SPI_SS_DISABLED  1
#define SPI_SAMPLE_AT_END 0x80  // Only for some parts
#define SPI_XMIT_L_TO_H  0x40   // Only for some parts
#define INT_SSP             0x8C08    // Used for ENABLE/DISABLE INTERRUPTS



#define INT_RDA             0x8C20    // Used for ENABLE/DISABLE INTERRUPTS
#define INT_TBE             0x8C10    // Used for ENABLE/DISABLE INTERRUPTS

///////////////////////////////////// Constants used for SETUP_ADC_PORTS()
#define ALL_ANALOG              0x80
#define ANALOG_RA3_REF          0x81
#define A_ANALOG                0x82
#define A_ANALOG_RA3_REF        0x83
#define RA0_RA1_RA3_ANALOG      0x84
#define RA0_RA1_ANALOG_RA3_REF  0x85
#define NO_ANALOGS              0x86

#define ANALOG_RA3_RA2_REF              0x88
#define ANALOG_NOT_RE1_RE2              0x89
#define ANALOG_NOT_RE1_RE2_REF_RA3      0x8A
#define ANALOG_NOT_RE1_RE2_REF_RA3_RA2  0x8B
#define A_ANALOG_RA3_RA2_REF            0x8C
#define RA0_RA1_ANALOG_RA3_RA2_REF      0x8D
#define RA0_ANALOG                      0x8E
#define RA0_ANALOG_RA3_RA2_REF          0x8F
///////////////////////////////////// Constants used for SETUP_ADC()
#define ADC_OFF                 0
#define ADC_CLOCK_DIV_2         1
#define ADC_CLOCK_DIV_8      0x41
#define ADC_CLOCK_DIV_32     0x81
#define ADC_CLOCK_INTERNAL   0xc1

#define ADC_DONE            0x8C40    // Used for ENABLE/DISABLE INTERRUPTS
```

```
#define INT_ADC             0x8C40    // Used for ENABLE/DISABLE INTERRUPTS

#list
```

C.4. Host computer calibration routine

Before running the Musical Trinkets program, the frequencies for each tag in the system had to be recorded so the program could distinguish between tags. This program was used for this calibration, and without being entirely too long, gives relevant information on interfacing a Windows PC with the tag reader board.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <conio.h>

const TOTALTAGS=20;
int maxes[TOTALTAGS];
int freqstart[TOTALTAGS], freqend[TOTALTAGS];
char tagnames[TOTALTAGS][30]={"red goblin","green goblin","blue goblin",
        "white ring", "red ring","yellow ring","green ring","blue ring",
        "distorting tag","dinosaur","porcupine","pikachu",
        "box1","box2","box3","pig","eye1", "eye2", "eye3", "cone(switch)"};
static HANDLE hCom;
char serialPort[30] = "COM2";
int i, j, k;
int data[256];
int pos=0;
int totalcount=10000;
char filename[80]="c:\\tags.siggraph\\tagcal.txt";
void comSetup();
void calibtag(int *start, int *end, int *max);
void savetofile();
int* read();

void main()
{
        // quick calibration-file-generating hack for the tags system


        // first, read in the current calibration-file
        ifstream infile(filename);
        if (infile == NULL) {
                cout << filename << " could not be loaded!  dropping to defaults...\n";
        }
        else {
                // load in current calibration
                infile >> serialPort;
                cout << "serial port " << serialPort << endl;
                for (i=0; i<TOTALTAGS; i++) {
                        infile >> freqstart[i] >> freqend[i] >> maxes[i];
                        cout << "tag " << i << ":" << freqstart[i] << " " << freqend[i] <<
        “ " << maxes[i] << '\n';
                }
        }
        comSetup();
        infile.close();
        cout << "Press key to continue...\n";
        cout.flush();
        _getch();

        int laststart=0, lastend=0, lastmax=0, lasttag=-1;
        int keeplooping=1;
        char input=0, currtag=0;
        while (keeplooping == 1) { // main menu loop
                while (_kbhit()) _getch();
                cout << "Current tag values (name, frequency range, amplitude):\n";
                for (i=0; i<TOTALTAGS; i++) { // print menu
                        cout << (char)((i<10)?'0'+i:('a'+(i-10)))
                                << ":" << tagnames[i] << " " << freqstart[i] << "-" <<
freqend[i] << " "
                                << maxes[i] << endl;
                }
                cout << "Press number/lowercase letter to calibrate tag, or S:save current
calibration,\n"
                        << "T:test tag (dump values), U:undo last tag calibrate, Q:quit\n";
                cout.flush();
                input=_getche();
                cout<<endl;
                switch(input) {
                        case 'Q':
                                keeplooping=0;
                                break;
                        case 'T':
                                while (!_kbhit()) {
                                        cout.flush();
                                        read();
                                        for (i=0; i<pos; i++) {
                                                cout << data[i] << " ";
                                        }
                                        cout << endl;
                                }
                                break;
                        case 'U':
                                if (lasttag!=-1) {
                                        freqstart[lasttag]=laststart;
                                        freqend[lasttag]=lastend;
                                        maxes[lasttag]=lastmax;
                                        lasttag=-1;
                                }
                                else cout << "Can't undo.\n";
                                break;
                        case 'S':
                                // calculate optimal frequencies and dump to file
                                //  - sort by freqstarts, then average freqstarts.
                                //  - if overlap, let lower tag take precedence
                                savetofile();
                                break;
                        default:
                                if (input >= '0' && input <= '9' && TOTALTAGS>input-'0') {
                                        currtag=input-'0';
                                }
                                else if (TOTALTAGS > 10 && (TOTALTAGS-10) > input-'a') {
                                        currtag=input-'a'+10;
                                }
                                else {
                                        cout << "invalid input.\n";
                                        break;
                                }
                                lasttag=currtag;
                                laststart=freqstart[currtag];
                                lastend=freqend[currtag];
                                lastmax=maxes[currtag];
                                calibtag(&freqstart[currtag], &freqend[currtag],
&maxes[currtag]);
                                break;
                }
        }
}

void calibtag(int *start, int *end, int *max) {
        int locounts[TOTALTAGS], hicounts[TOTALTAGS], sums[TOTALTAGS], middles[TOTALTAGS];
        *start=10000;
        *end=0;
        *max=0;
        while (!_kbhit()) {
                read();
                if ((pos-4)/4 <= 0) {
                        continue;
                }
                for (i=0; i<pos; i+=4)
                {
                        locounts[i/4]=data[i];
                        hicounts[i/4]=data[i+1];
                        sums[i/4]=(data[i+2]<<8) + data[i+3];
                        middles[i/4]=(locounts[i/4]+hicounts[i/4])/2;
                }
                totalcount=(data[pos-4]<<8)+data[pos-3];
                if (*start>locounts[0]) *start=locounts[0];
                if (*end<hicounts[0]) *end=hicounts[0];
                if (sums[0]>*max) *max=sums[0];
                cout << (*start) << "-" << (*end) << " " << (*max) << endl;
        }
}

void savetofile() {
        int order[TOTALTAGS], realstart[TOTALTAGS], realend[TOTALTAGS];
        int temp;
        for (i=0; i<TOTALTAGS; i++)
                order[i]=i;
        for (j=0; j<TOTALTAGS; j++) {
                for (i=0; i<TOTALTAGS-1; i++) {
                        if (freqstart[order[i]]>freqstart[order[i+1]]) {
                                temp=order[i];
                                order[i]=order[i+1];
                                order[i+1]=temp;
                        }
                }
        }
        realstart[order[0]] = 0;
        for (i=0; i<TOTALTAGS-1; i++) {
                if (freqend[order[i]]>=freqstart[order[i+1]]) {
```

```
			cout << "warning!  overlap between " << tagnames[order[i]] << " and 
" << 
				tagnames[order[i+1]] << endl;
			realend[order[i]]=freqend[order[i]];
			realstart[order[i+1]]=freqend[order[i]]+1;
		}
		else {
			realend[order[i]]=(freqend[order[i]]+freqstart[order[i+1]])/2;
			realstart[order[i+1]]=realend[order[i]]+1;
		}
	}
	realend[order[TOTALTAGS-1]] = totalcount;

	ofstream outfile(filename);
	outfile << serialPort << endl;
	for (i=0; i<TOTALTAGS; i++) {
		outfile << realstart[i] << ' ' << realend[i] << ' ' << maxes[i] << endl;
	}
	outfile.close();
	cout << "file " << filename << "written.";
	_getche();
}

int* read() {
	DWORD bytesread;
	unsigned char buffer[256];

	pos=0;
	while (1)
	{
//		Sleep(1);
		ReadFile(hCom, buffer, 1, &bytesread, NULL);
//		cout << "read" << bytesread;
		cout.flush();

		if (bytesread)
		{
//			cout << "read";
//			cout.flush();
			for (int i=0; i<bytesread; i++)
			{
				data[pos++]=buffer[i];
				if (pos>250) pos=0;
				if (data[pos-1]==255 && data[pos-2]==255)
				{
//					cout << endl << pos << endl;
					return data; // data in data, pos in pos (globals)
				}
			}
		}

	}
}

static
void
comSetup()
{
	DCB dcb;
	DWORD dwError;
	BOOL fSuccess;
```

```
        hCom = CreateFile(serialPort,
                GENERIC_READ | GENERIC_WRITE,
                0,    /* comm devices must be opened w/exclusive-access */
                NULL, /* no security attrs */
                OPEN_EXISTING, /* comm devices must use OPEN_EXISTING */
                0,    /* not overlapped I/O */
                NULL  /* hTemplate must be NULL for comm devices */
                );

        cout << "Opened serial port " << serialPort << " " << hCom;
        if (hCom == INVALID_HANDLE_VALUE) {
                dwError = GetLastError();

                /* handle error */
//              r->ew.w("Error opening %s",serialPort);
                return;
        }

        /*
        * Omit the call to SetupComm to use the default queue sizes.
        * Get the current configuration.
        */

        fSuccess = GetCommState(hCom, &dcb);

        if (!fSuccess) {
                /* Handle the error. */
//              r->ew.w("Error getting the comm state");
                return;
        }

        /* Fill in the DCB: baud=19200, 8 data bits, no parity, 1 stop bit. */

        dcb.BaudRate = 19200;
        dcb.ByteSize = 8;
        dcb.Parity = NOPARITY;
        dcb.StopBits = ONESTOPBIT;

        fSuccess = SetCommState(hCom, &dcb);

        if (!fSuccess) {
            /* Handle the error. */
//              r->ew.w("Error setting the comm state");
                return;
        }

        COMMTIMEOUTS cto;
        cto.ReadIntervalTimeout = 4;
    cto.ReadTotalTimeoutMultiplier = 10;
    cto.ReadTotalTimeoutConstant = 100;
    cto.WriteTotalTimeoutMultiplier = 10;
    cto.WriteTotalTimeoutConstant = 100;
        if (!SetCommTimeouts(hCom, &cto)){
//              r->ew.w("Unable to set proper timeouts");
        }
}
```