NATIONAL RADIO ASTRONOMY OBSERVATORY
GREEN BANK, WEST VIRGINIA

ELECTRONICS DIVISION TECHNICAL NOTE NO. 186

TITLE: **A "Clean" 8-bit ADC and Data Acquisition Card**

AUTHOR: **D. Anish Roshi**

DATE: **July 24, 2001**

Internal Technical Report

A "Clean" 8-bit ADC and Data Acquisition Card

D. Anish Roshi
Date: 24-7-01

1 Introduction

The outputs of most commercially available Analog-to-Digital converters (ADCs) have spurious components in addition to the digitized form of the desired signal. These spurious components usually appear as narrow band features in the spectrum of the digitized waveform. The spurious components at the output of the ADC limit the performance of the converter for many applications – for example RFI mitigation. The spurious components are due to noise, generated in the associated digital circuits ("digital noise"), coupling to the analog input. The most common reasons for such noise coupling are (a) poor isolation of analog and digital grounds (b) insufficient power supply filtering and (c) poor design of associated digital circuit, which results in excessive ground bounces.

This report gives the design of an ADC, designed for eliminating any spurious components at its output, and a data acquisition system (DAS). The results of the performance tests done on the ADC are given in Section 4. The results show that no narrow band spurious components are present in the spectrum of the ADC output when the data is integrated for 34 mts. The ADC and DAS are tested up to a maximum clock frequency of 70 MHz.

2 Circuit

2.1 ADC

The ADC is designed using the IC ADS831 (Burr-Brown product). The attractive features of this IC are the optional balanced analog input and the low-voltage (3.3 V) operation of the digital circuit. The balanced input is immune to any common mode pickups. The low-voltage operation of the digital part of the IC helps in reducing digital noise. The ADS831 has an 8 bit quantizer.

The schematic of the ADC is given in Appendix A. Four ADS831 were used in the design. The analog input is balanced using a 1:2 transformer TT1-6. The analog part of ADS831 operates at 5 V. We designed the circuit such that the digital part of the ADS831 operates at 3.3 V. The outputs of the ADCs are connected to a low-voltage buffer. The other design features include:

1. Ferrite cores based filters used for the power supply, which minimize noise coupling through power connections.

2. Ferrite core based filters used at the digital power pin of ADS831 for better decoupling.

3. Low-voltage buffers (74LVTH162244) with an internal series termination resistor of 22 Ω were used for the design. This helps in reducing the reflections in the interconnection cable thus minimizing the ground bounces.

4. For the PCB design, the analog and digital power and ground planes are separated. The ground planes are interconnected near ADS831 to provide the signal return paths.

5. We followed the "20-H rule" (Montrose 1996) where ever the analog and digital power planes are at proximity.

2.2 DAS

The data acquisition system is designed to acquire data from the ADCs using a PC. It consists of 128 KByte FIFOs for each of the four ADCs (see Appendix B). The data from the ADCs can be acquired in "burst mode" using the DAS (see Section. 3). The DAS is connected to the PC through a PCI based I/O card (ADLink's PCI-7200). Data from the DAS is transfered to the PCI-7200 using the handshake mode of I/O card. The design features of the DAS are the following.

1. Care is taken to isolate DAS ground from the ADC ground and also from the PC ground. The ADC ground and DAS ground are "isolated" using differential line drives (MC100LVELT22) and receivers (MC100LVELT23). The PC ground is optically isolated from the DAS ground. These ground isolations are provided to minimize the digital noise coupling to the ADC circuit.

2. A state machine is implemented in a GAL (GAL22LV10) for generating the handshake signal for the data transfer from the DAS to PCI-7200. The timing diagram for the data transfer is given in Fig. 2. The codes used for programming the GALs are given in Appendix C.

3 Burst Mode Data Acquisition

In the burst mode of data acquisition, on power ON or reset, the DAS waits till the FIFOs become almost full (AF flag). On the high to low transition of AF flag the DAS sends a data request signal (BDI_REQ) to the PCI-7200. PCI-7200 acknowledges a successful data transfer with the I_ACK signal. The data transfer continues in this fashion if the FIFO is not full. If the FIFO is filled, the FIFO is disabled from writing any new data samples till the data in the FIFO is completely transfered to the PC. A new burst of data will be written into the FIFO on FIFO empty and the burst mode cycle continues. The code of the software used for acquiring data using PCI-7200 in the burst mode is given in Appendix D.

4 Results of the ADC Performance test

A few tests were conducted to evaluate the performance of the ADC and the DAS. For all these tests we interconnected the different grounds at the power supply.

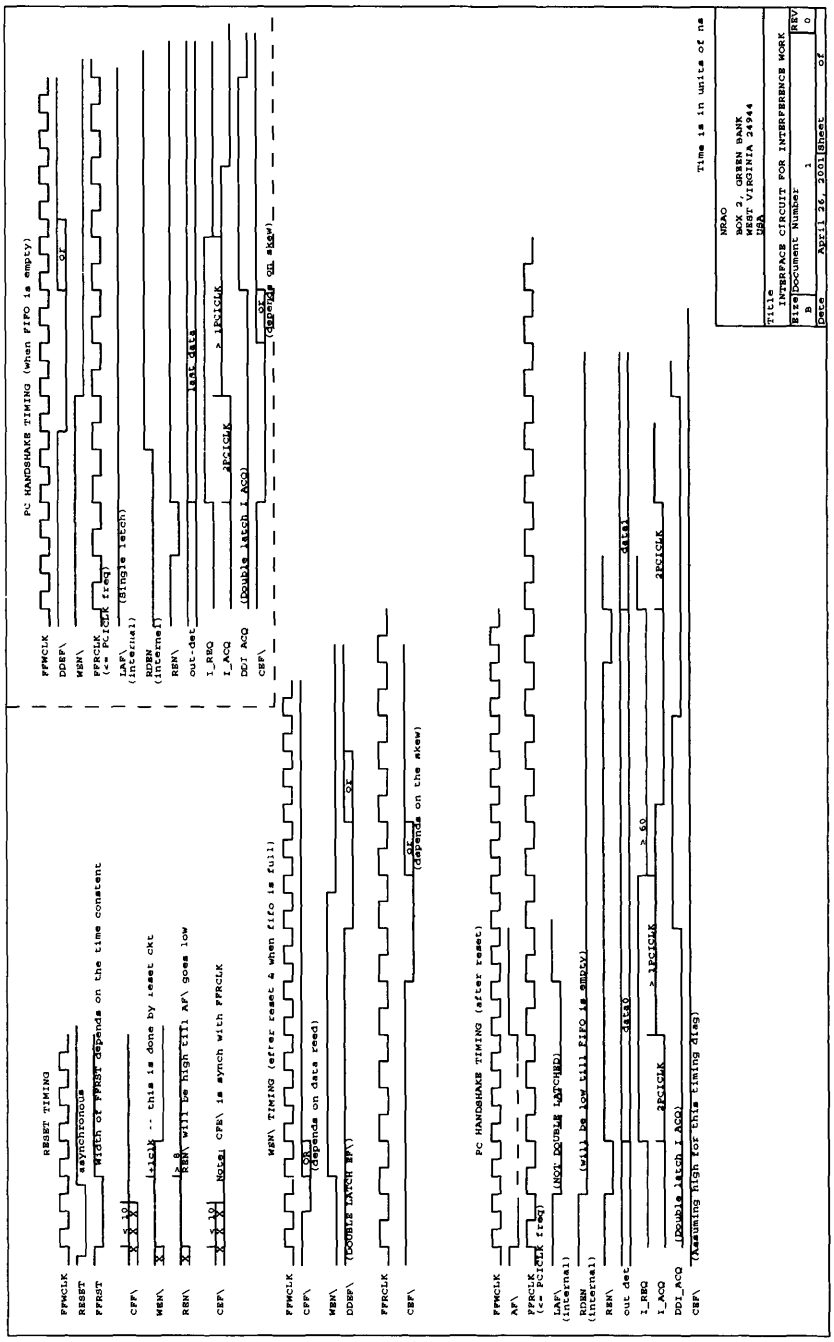

Figure 1: The timing diagram for the data transfer from DAS to PCI-7200. The timing on reset or power ON is given on top left.

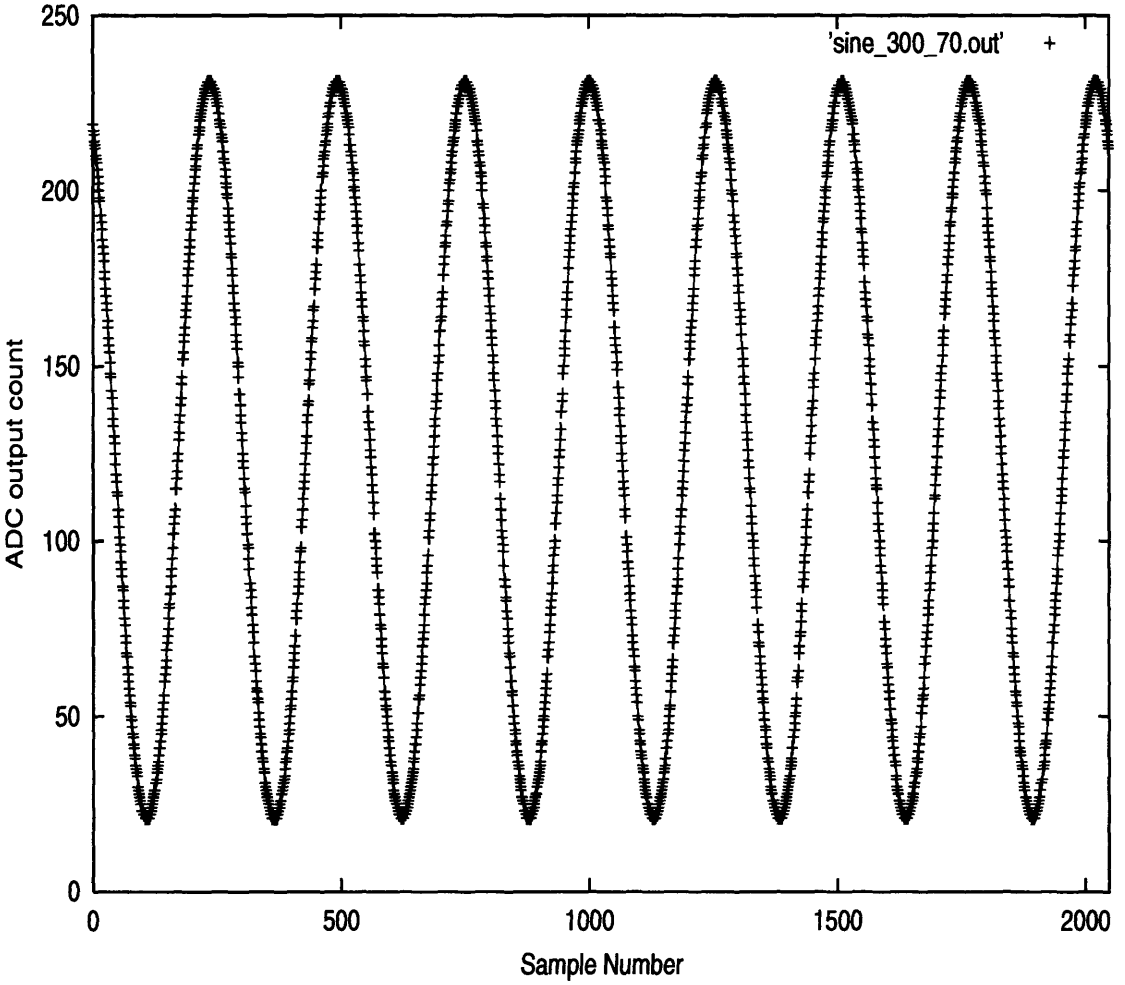

Figure 2: ADC output as a function of sample number. The input CW signal frequency is 300 KHz and the sampling frequency is 70 MHz. The bits of the ADC and the DAS are changing as expected. The fluctuations at the least significant bit may be due to noise.

1. **Sine wave test :** A 300 KHz CW (from Hp 3312A Function generator) with an appropriate amplitude to make the ADC output change over almost the full scale (0 to 255) is fed at the input of one of the ADCs (U1 in the circuit schematic). No input is connected to other ADCs. The ADC and the DAS are operated at a clock frequency of 70 MHz. We used Gigatronics 6100 synthesized signal generator for generating the 70 MHz clock frequency. Fig. 1 shows that all the bits change as expected. A one bit jitter can be seen which could be due to noise fluctuations.

2. **Long term integration of ADC output :** A noise source is connected to one of the ADCs. The bandwidth of the noise is limited to about 12 MHz and the power level is adjusted such that the ADC output range of 0 to 255 represents about 5 times the RMS fluctuations. We reduced the clock frequency of ADC to 30 MHz for this test since the data acquisition is not very efficient. At this clock frequency

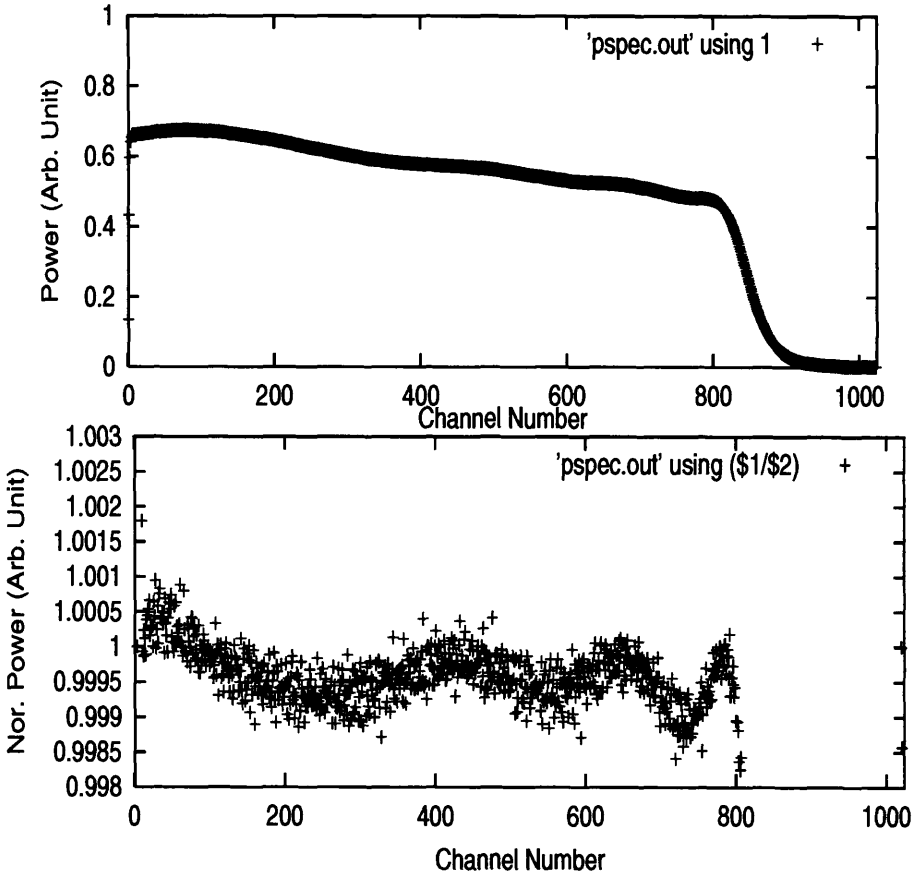

Figure 3: 34 mts integrated bandshape of the noise source (top) and spectrum after removing the bandshape (bottom). No spurious narrow band signals are present at 3 sigma level, which corresponds to a spurious level of < -32 dB relative to the noise power.

the efficiency of data acquisition through the PC is about 1%, which means that to accumulate 1 sec of data the acquisition system takes about 100 sec. The sampling frequency is again generated using Gigatronics 6100 synthesized signal generator. In this mode, we accumulated the output spectrum for about 34 mts (see Fig. 2). The integrated output spectrum is divided by a smoothed (9 point boxcar) version of the same spectrum. This essentially takes out the bandshape of the noise source. No narrow band spurious signal is present at the output data at -32 dB (corresponding to 3 sigma) level relative to noise power. The rms noise in the bandshape corrected spectrum is close to the theoretically expected value.

3. **Clock jitter :** A CW signal of 16 MHz (from Hp 83620A synthesized sweep generator) is fed as input of one of the ADCs (U1). The sampling frequency was set to 64 MHz (from Hp 837123 Synthesized CW generator) and the input signal is locked to the

sampling frequency. This will ensure that the spectrum of the CW signal is centered at channel 512 after a 2048 point Fourier transform. The spectrum of the ADC output, integrated over 40 msec, (see Fig. 3) shows that the power at an offset of 100 KHz (about 3 channels) from the channel at which the CW is present is about 3 dB more than that measured with the spectrum analyzer (ADVANTEST R3261A). The resolution bandwidth of the spectrum analyzer was set to 30 KHz for this measurement, which is similar to the spectral resolution of the power spectrum. This excess power could be due to the jitter in the sampling clock. Further study of the clock jitter will be done when the need arises.

4. **Cross coupling :** The cross coupling between the ADCs is measured by feeding a CW signal at 10 MHz to ADC U3 and 16 MHz to ADC U1. Comparison of the spectrum of the ADC (U1) output with and without the 10 MHz CW signal in the second ADC (U3) (see Fig. 4) shows that any cross coupling of the 10 MHz signal is at a level < -65 dB relative to the 16 MHz signal. We haven't checked the cross coupling at other frequencies.

5 Acknowledgment

I thank Rick Fisher for the suggestion that I build a "Clean" ADC for RFI mitigation work and for the many fruitful discussions I had with him. I am grateful to Rich Bradley, Chuck Broadwell and Ray Escoffier for the useful discussion during the design and construction of this project. I thank Gary Anderson, Ann Wester, Galen Watts, Steve White, Mike Stennes, Bill Shank and all technicians at the digital laboratory for providing the necessary tools and accessories for building and testing the circuit.

6 Reference

Montrose, M. I., 1996, "Printed Circuit Board Design Techniques for EMC Compliance", IEEE Press Series on Electronics Technology, IEEE, Inc., New York.

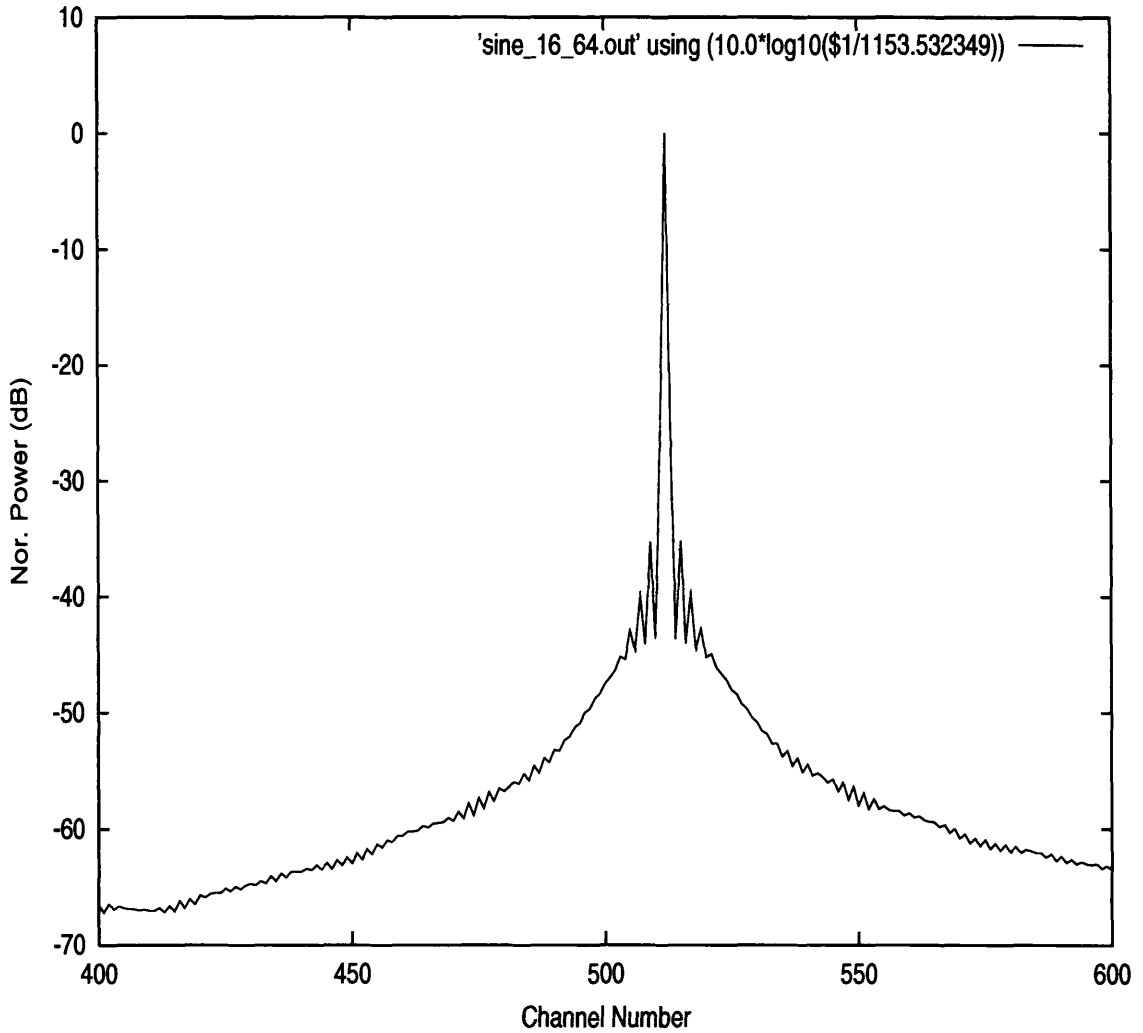

Figure 4: 40 msec integrated spectrum of a 16 MHz CW signal obtained from the ADC output. The sampling frequency was set to 64 MHz to center the CW signal at channel 512. Power at channel 515 and 509 is about 3 dB more than those measured with a spectrum analyzer with similar spectral resolution. This excess power may be due to clock jitter and needs further investigation.

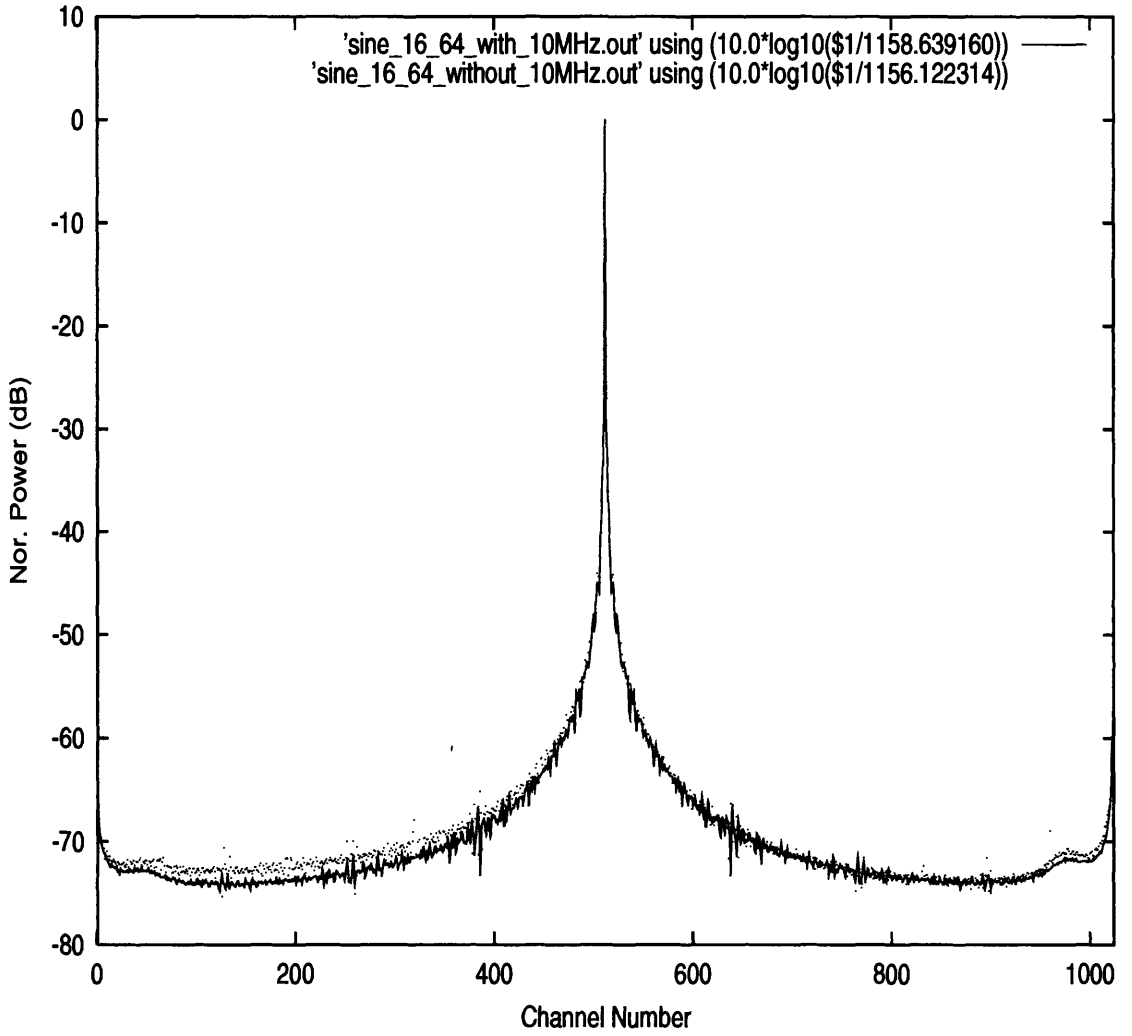

Figure 5: 40 msec integrated spectrum of a 16 MHz CW signal from one of the ADCs with (solid line) and without (dots) a 10 MHz CW signal at a second ADC. The cross coupling of the 10 MHz signal is at level < -65 dB relative to the 16 MHz CW power.

Appendix A
ADC Circuit Schematic

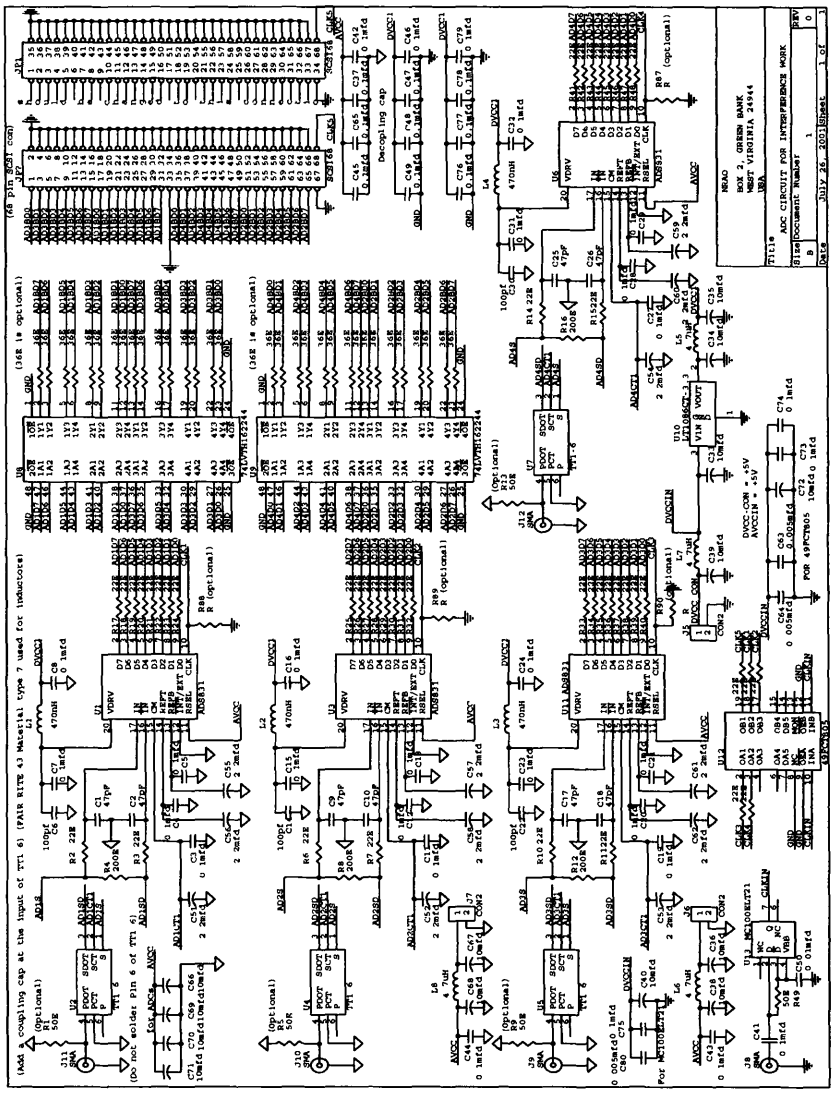

Figure 6: Schematic of the ADC circuit.

Appendix B
DAS Circuit Schematic

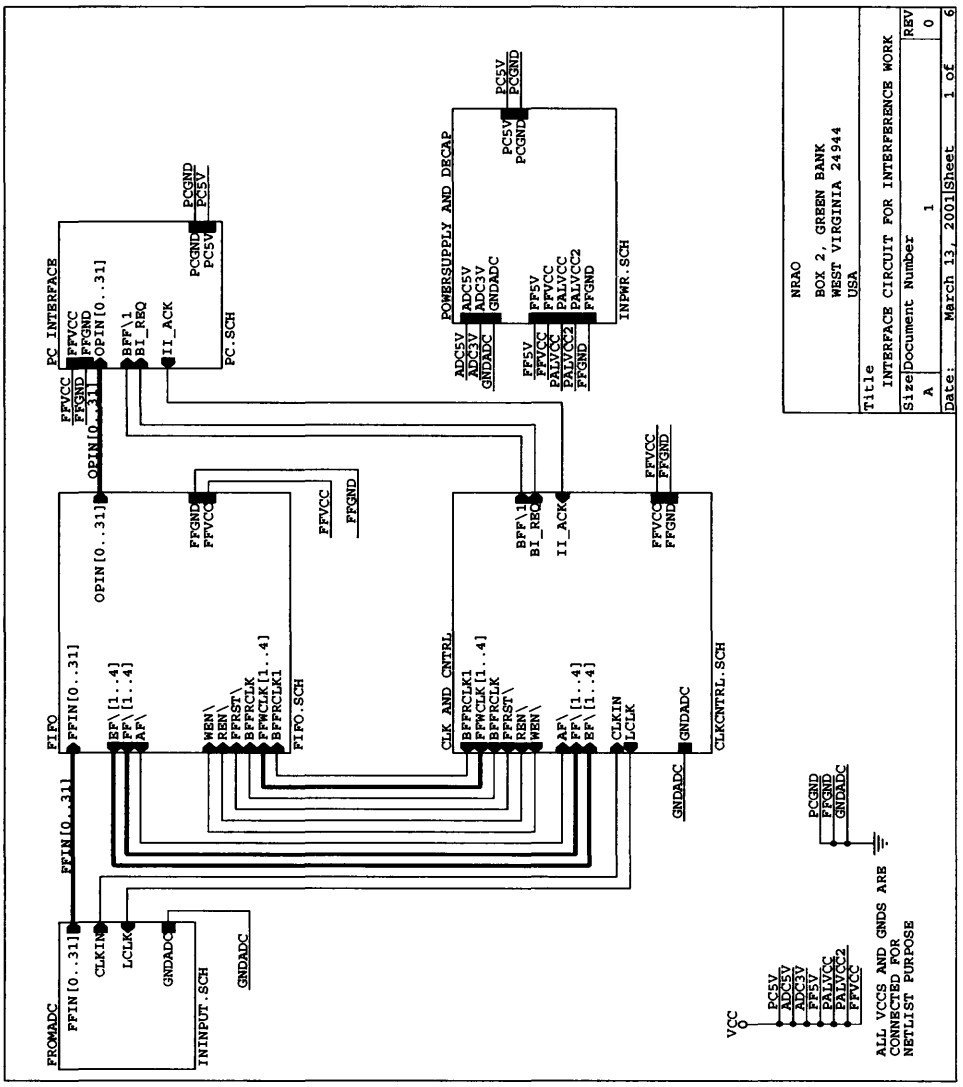

Figure 7: Schematic of the DAS circuit.

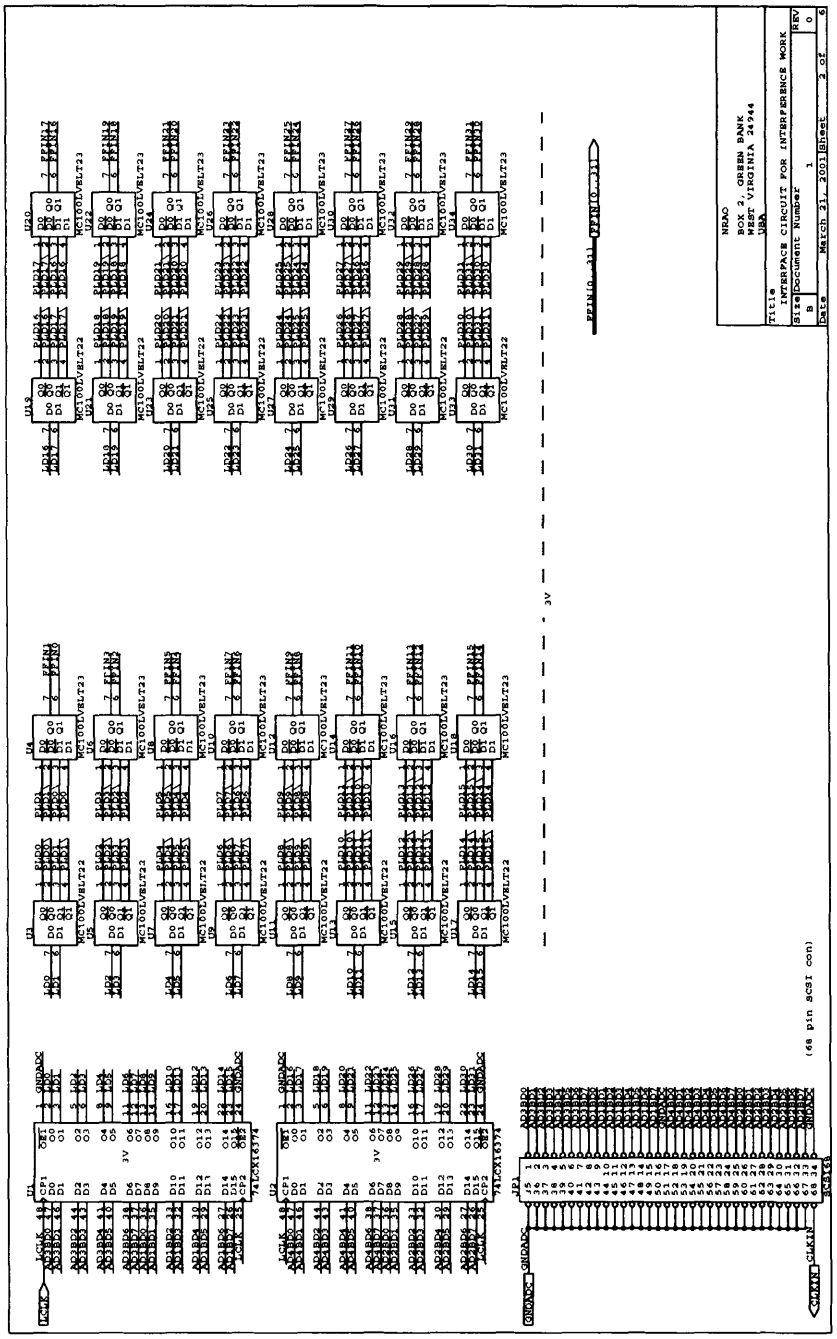

Figure 8: Schematic of the DAS circuit (continued).

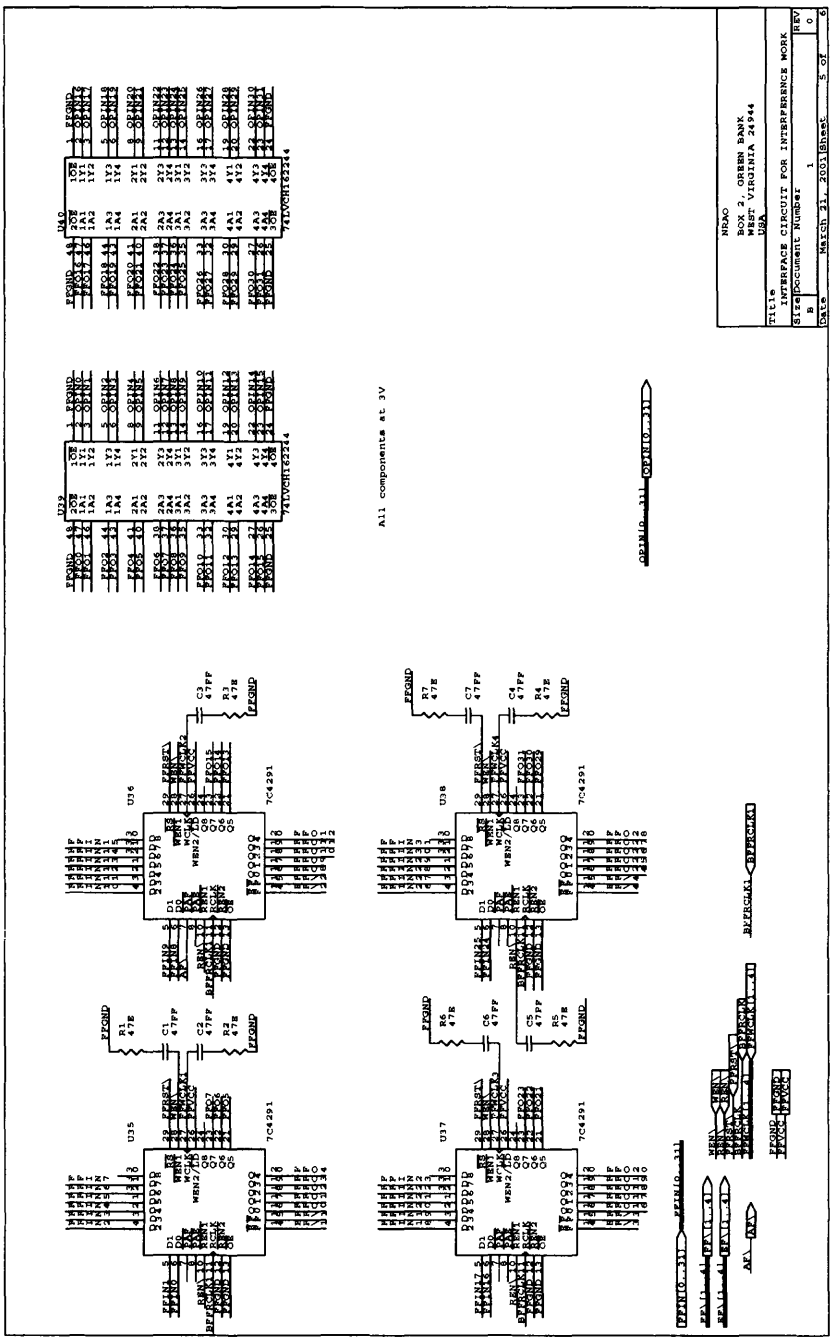

Figure 9: Schematic of the DAS circuit (continued).

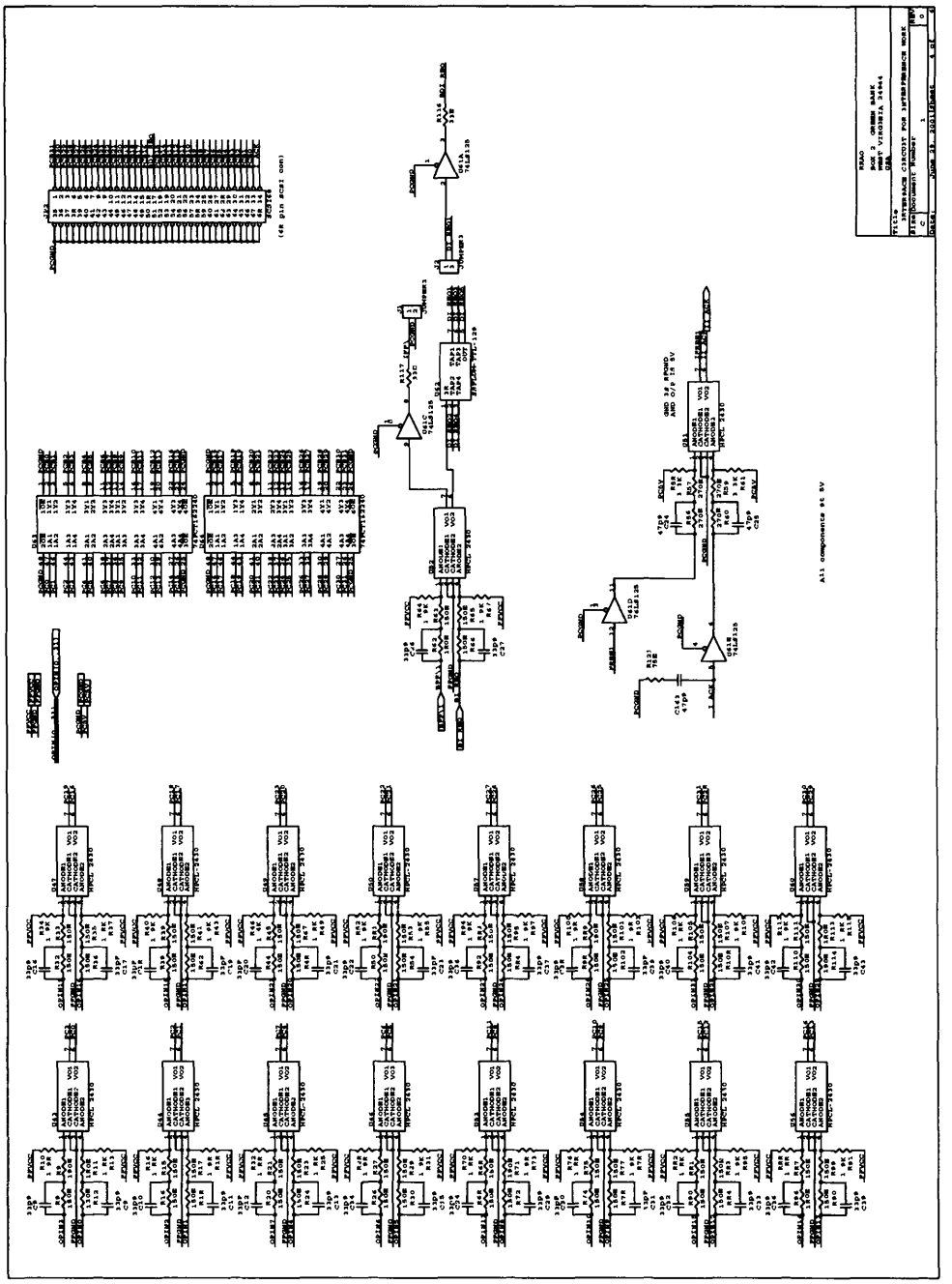

Figure 10: Schematic of the DAS circuit (continued).

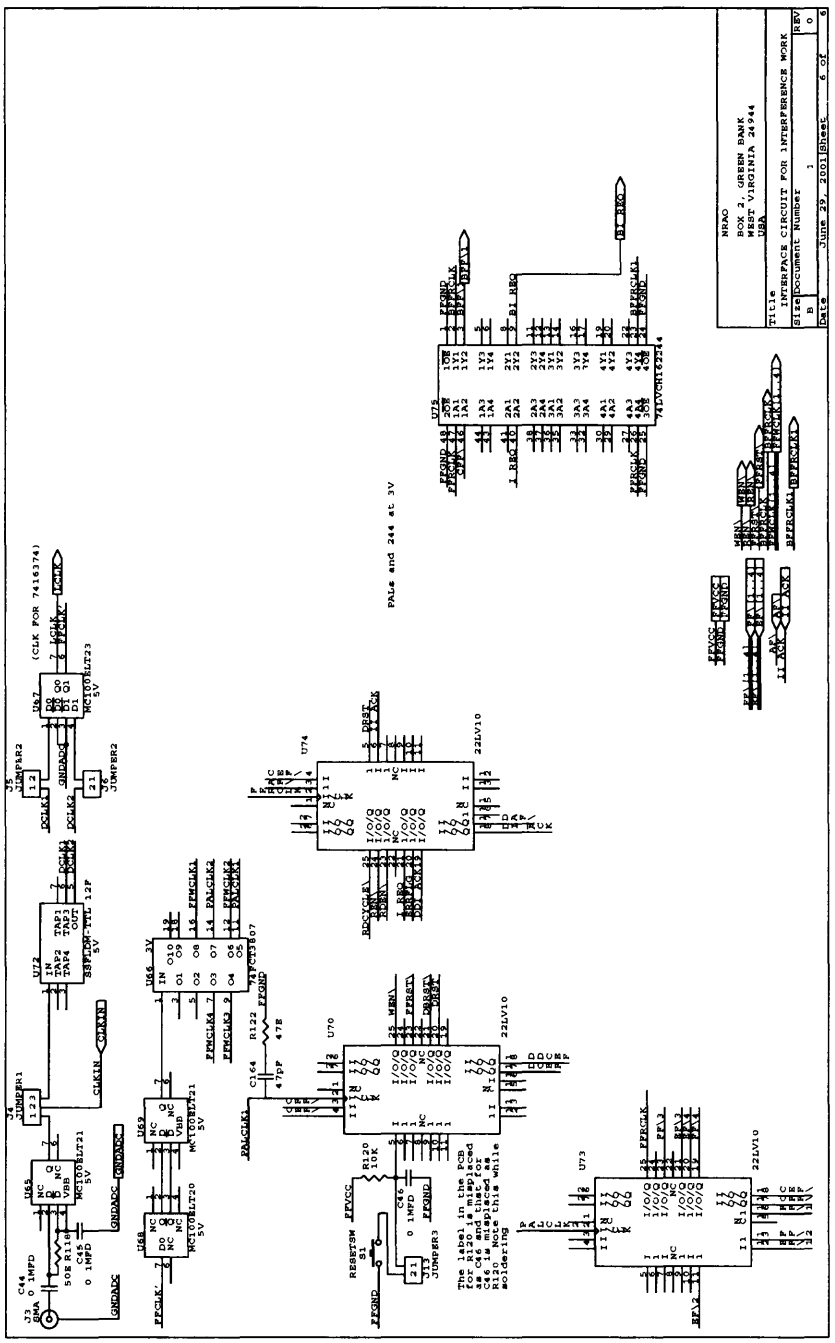

Figure 11: Schematic of the DAS circuit (continued).

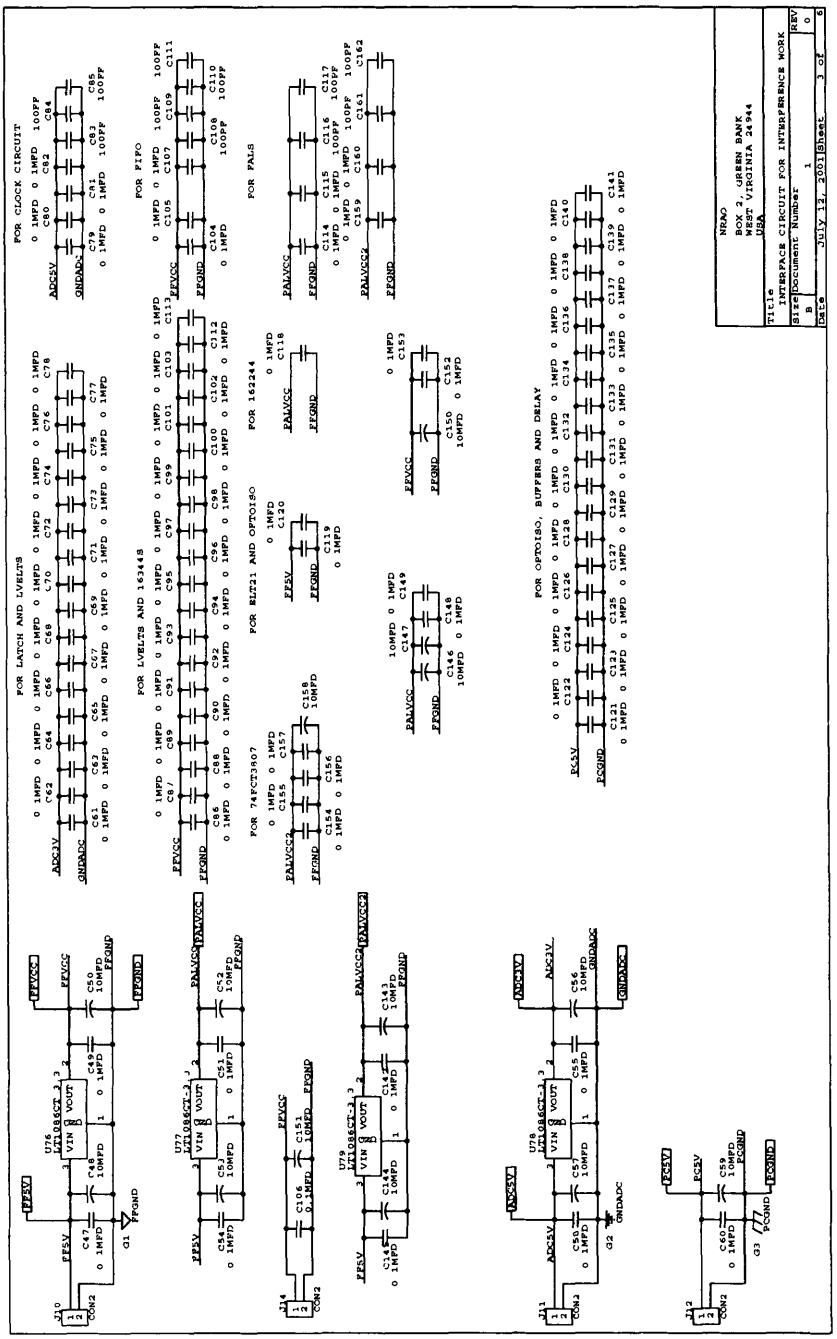

Figure 12: Schematic of the DAS circuit (continued).

Appendix C
Codes used for programming the GALs

```
/* Code used for programing U70 GAL in DAS circuit */

Name      rst_state;
Partno    u70;
Date      04/25/01;
Revision  02;
Designer  Anish;
Company   NRAO, GB;
Assembly  PCinterface;
Location  U70;
Device    g22v10lcc;

/**  Inputs  **/

Pin 2       = palclk1;          /* clock */
Pin 5       = RST;              /* Power on reset  */
Pin 3       = CFF_;             /* Combined FIFO full  */
Pin 4       = CEF_;             /* Combined FIFO empty */

/**  Outputs  **/

Pin 23 = FFRST_;      /* Reg reset for FIFO */
Pin 21 = DBRST_;      /* Reg reset for other pals */
Pin 25 = WEN_;        /* FIFO Write enable     */
Pin 26 = DRST_;
Pin 17 = DCEF_;
Pin 18 = DDCEF_;
Pin 19 = RST_ST;


/** Logic Equations for reset latch **/

DRST_.d = RST;
DRST_.OE = 'b'1;
DRST_.AR = 'b'0;
DRST_.SP = 'b'0;

/* Equation for DBRST_ */

DBRST_ = FFRST_;
```

```
/* Logic equation for CEF\ double latch */

DCEF_.d = CEF_;
DCEF_.OE = 'b'1;
DCEF_.SP = 'b'0;
DCEF_.AR = 'b'0;

DDCEF_.d = DCEF_;
DDCEF_.OE = 'b'1;
DDCEF_.SP = 'b'0;
DDCEF_.AR = 'b'0;

/** Declarations and Intermediate Variable Definitions **/

field WEN_ST = [RST_ST, FFRST_, WEN_];  /* declare state bit field */

/* define counter states */

$define S0 'b'000  /* Not used */
$define S1 'b'001  /* Reset state */
$define S2 'b'010  /* Error state */
$define S3 'b'011  /* State to extent WEN_ = 1 after Reset */
$define S4 'b'100  /* Not used */
$define S5 'b'101  /* Not used */
$define S6 'b'110  /* FIFO Write/read enable state */
$define S7 'b'111  /* FIFO Write disable but read enable state */

/** For State bits **/

WEN_ST.OE = 'b'111;
WEN_ST.AR = 'b'000;
WEN_ST.SP = 'b'000;

Sequenced WEN_ST {
present S1   if !DRST_          next S1;
             if DRST_           next S3;
present S3   if DRST_ & CFF_    next S6;
             if DRST_ & !CFF_   next S2; /* Error state */
             if !DRST_          next S1;
present S6   if DRST_ & CFF_    next S6;
             if DRST_ & !CFF_   next S7;
             if !DRST_          next S1;
present S7   if DRST_ & DDCEF_  next S7;
             if DRST_ & !DDCEF_ & CFF_  next S6;
             if DRST_ & !DDCEF_ & !CFF_ next S2; /* Error state */
```

```
                if !DRST_           next S1;
present S2      if !DRST_           next S1;
                if DRST_            next S2;
present S0      if !DRST_           next S1;
                if DRST_            next S0;
present S4      if !DRST_           next S1;
                if DRST_            next S4;
present S5      if !DRST_           next S1;
                if DRST_            next S5;
}
```

```
/* Code used for programing U73 GAL in DAS circuit */

Name      CLKDIV;
Partno    U73;
Date      02/10/01;
Revision  00;
Designer  Anish;
Company   NRAO, GB;
Assembly  PCinterface;
Location  U73;
Device    G22v10lcc;

/**  Inputs  **/

Pin 2       = palclk;          /* Counter clock       */
Pin 16      = FF_1;
Pin 13      = FF_2;
Pin 23      = FF_3;
Pin 19      = FF_4;
Pin 12      = EF_1;
Pin 11      = EF_2;
Pin 21      = EF_3;
Pin 20      = EF_4;

/**  Outputs  **/

Pin [25..27] = [Q0..2];         /* Counter outputs    */
Pin 17  = CFF_;
Pin 18  = CEF_;

/** Declarations and Intermediate Variable Definitions **/

field count = [Q2..0];          /* declare counter bit field */
$define S0 'b'000               /* define counter states */
$define S1 'b'001
$define S2 'b'010
$define S3 'b'011
$define S4 'b'100
$define S5 'b'101
$define S6 'b'110
$define S7 'b'111

/** Logic Equations **/
```

```
count.OE = 'b'111;
count.AR = 'b'000;
count.SP = 'b'000;

Sequenced count {              /* free running counter */

present S0   next S1;
present S1   next S2;
present S2   next S3;
present S3   next S4;
present S4   next S5;
present S5   next S6;
present S6   next S7;
present S7   next S0;

}

/* Equation for CEF\ and CFF\ */

CFF_ = FF_1 & FF_2 & FF_3 & FF_4;
CEF_ = EF_1 & EF_2 & EF_3 & EF_4;
```

```
/* Code used for programing U74 GAL in DAS circuit */

Name      PCINTERF;
Partno    U74;
Date      02/13/01;
Revision  00;
Designer  Anish;
Company   NRAO;
Assembly  PCinterface;
Location  U74;
Device    g22v10lcc;


/**  Inputs  **/

Pin 2        = FFRCLK;          /* FIFO read clock              */
Pin 3        = AF_;             /* Almost full flag             */
Pin 4        = CEF_;            /* Combined empty flag          */
Pin 5        = DRST;            /* Power on reset after first latch */
Pin 6        = II_ACK;          /* PC acknowledgement           */

/**  Outputs  **/

Pin 24       = REN_;            /* FIFO Read enable             */
Pin 21       = !I_REQ;          /* Request to PC; complemented to */
                          /* take care of the optoisolator inversion */

Pin 23       = RDEN_;           /* Read enable (internal)       */
Pin 25       = RDCYCLE_;        /* Read cycle (internal)        */
Pin 20       = ERRFLG;          /* Error FLAG                   */
Pin 26       = LERRFLG;         /* Latched the ERRFLG once it occurs */

Pin 17       = DAF_;
Pin 18       = DI_ACK;
Pin 19       = DDI_ACK;

/** Declarations and Intermediate Variable Definitions **/

field rd_state = [RDEN_,RDCYCLE_,REN_,I_REQ]; /* declare state variables */
$define S0 'b'0000              /* define states */
$define S1 'b'0001
$define S2 'b'0010
$define S3 'b'0011
$define S4 'b'0100
```

```
$define S5 'b'0101
$define S6 'b'0110
$define S7 'b'0111
$define S8 'b'1000
$define S9 'b'1001
$define S10 'b'1010
$define S11 'b'1011
$define S12 'b'1100
$define S13 'b'1101
$define S14 'b'1110
$define S15 'b'1111

/** Logic Equations **/

DAF_.d = AF_;              /* Single latch AF\ */
DAF_.OE = 'b'1;
DAF_.ar = 'b'0;
DAF_.sp = 'b'0;

DI_ACK.d = II_ACK;         /*Double latch II_ACK */
DI_ACK.oe = 'b'1;
DI_ACK.ar = 'b'0;
DI_ACK.sp = 'b'0;

DDI_ACK.d = !DI_ACK;     /*Inverted because of optoisolator */
DDI_ACK.oe = 'b'1;
DDI_ACK.ar = 'b'0;
DDI_ACK.sp = 'b'0;

LERRFLG.d = (LERRFLG # ERRFLG) & DRST;
LERRFLG.OE = 'b'1;
LERRFLG.ar = 'b'0;
LERRFLG.sp = 'b'0;

rd_state.OE = 'b'1111;
rd_state.ar = 'b'0000;
rd_state.sp = 'b'0000;

ERRFLG.OE = 'b'1;
ERRFLG.ar = 'b'0;
ERRFLG.sp = 'b'0;

Sequenced rd_state {                          /* State machine */
present S14   if (DRST & !DAF_ & !DDI_ACK)      next S4;
              if (DRST & !DAF_ & DDI_ACK)       next S2 OUT ERRFLG;
```

```
                if (DRST & DAF_)                    next S14;
                if !DRST                            next S14 OUT !ERRFLG;
present S4      if DRST                             next S3;
                if !DRST                            next S14 OUT !ERRFLG;
present S3      if (DRST & !CEF_)                   next S11;
                if (DRST & DDI_ACK & CEF_ )         next S2;
                if (DRST & !DDI_ACK & CEF_)         next S3;
                if !DRST                            next S14 OUT !ERRFLG;
present S11     if (DRST & DDI_ACK & DAF_)          next S10;
                if (DRST & !DDI_ACK & !DAF_)        next S3;
                if (DRST & DDI_ACK & !DAF_)         next S2;
                if (DRST & !DDI_ACK & DAF_)         next S11;
                if !DRST                            next S14 OUT !ERRFLG;
present S2      if (DRST & !DDI_ACK & CEF_)         next S4;
                if (DRST & DDI_ACK  & CEF_)         next S2;
                if (DRST & DDI_ACK  & !CEF_)        next S2  OUT ERRFLG;
                if (DRST & !DDI_ACK & !CEF_)        next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S10     if (DRST & DAF_ & !DDI_ACK)         next S14;
                if (DRST & !DAF_ & DDI_ACK)         next S2;
                if (DRST & DAF_ & DDI_ACK)          next S10;
                if (DRST & !DAF_ & !DDI_ACK)        next S4;
                if !DRST                            next S14 OUT !ERRFLG;
present S15     if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S13     if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S12     if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S9      if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S8      if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S5      if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S1      if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S0      if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S6      if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
present S7      if DRST                             next S14 OUT ERRFLG;
                if !DRST                            next S14 OUT !ERRFLG;
}
```

Appendix D
Software code used for data acquisition

```
// Program to get the ADC output

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "dask.h"

U32 DoBuf[131072]; /* 128 K buffer */
char *file_name="adc.out"; /* output file */
U32 read_count=131072;

main()
{
    I16 card, err, card_num;
    U32 count, i, j;
    FILE *outfile;

    printf("Please input a card number: ");
    scanf(" %d", &card_num);

    if ((card=Register_Card (PCI_7200, card_num)) <0 ) {
        printf("Register_Card error=%d\n", card);
        exit(1);
    }

// Configure the PCI-7200 in handshake mode

    err = DI_7200_Config(card, TRIG_HANDSHAKE, DI_NOWAITING,
                         DI_TRIG_FALLING, IREQ_RISING);
    if (err!=0) {
       printf("DI_7200_Config error=%d", err);
       exit(1);
    }


    outfile = fopen(file_name, "w");

    for (i=0; i<10; i++)
    {
      // Read 128 K words from the FIFOs
      err = DI_ContReadPort(card, 0, DoBuf, read_count,
```

```
                        CLKSRC_EXT_SampRate, SYNCH_OP);

      if (err!=0) {
          printf("DI_ContReadPort error=%d", err);
          exit(1);
      }

      //Write the data from ADC U1 to the file
      for (j=0; j<(128*1024); j++)
        fprintf(outfile, "%d\n",
               ((unsigned int)((DoBuf[j] & 0x0000ff00) >> 8)));

      printf("%d \n",i);

    }

    fclose(outfile);

    DI_AsyncClear(card, &count);

    DO_AsyncClear(card, &count);

    Release_Card(card);

}
```

```
/* This program integrates the power spectrum          */
/* of the ADC output for a specified number of samples */
/* and append to a file                                */

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "dask.h"
#include <math.h>

#define SIZE_FFT  2048 /* FFT length */
U32 DoBuf[SIZE_FFT]; /* 2k buffer */
char *file_name="adc.out"; /* output file name */
U32 read_count= SIZE_FFT;



/* This routine computes a fast Fourier transform and

 * replaces the input data arguments by the output spectrum.

 * It assumes that N is a power of 2 and that 'data_real' and

 * 'data_imag' have N elements.

 * FFT routine is from Rick.

 */

#define   TWOPI     6.283185307

int bit_reverse ( int num_bits, int word )
{

    int result;
    int test_bit;
    int i;

    result = 0;
    test_bit = 1;

    for (i = 0; i < num_bits; i++) {
result <<= 1;
```

```
if (test_bit & word) result |= 1;
test_bit <<= 1;
    }

    return result;
}



void FFT ( int N, float *data_real, float *data_imag )
{

     static int last_N = 0;   /* If N is the same as the last time this
       * routine was called, don't recalculate the
       * W coefficients */

     static int num_stages;   /* number of butterfly stages */
     static int half_N;       /* N / 2 */
     static int *bit_reversed_index = 0;

     int bix;                 /* butterfly index */
     int cix;                 /* coefficient index */
     int cix_incr;            /* coefficient index increment */
     int dixA;                /* data index, A input */
     int dixB;                /* data index, B input */
     int stage;               /* butterfly stage */
     int skip_len;            /* data index skip distance */
     int skip_pt;             /* data index skip point */
     int num_bits;       /* number of bits in data index */

     static float *coef_real = 0;  /* W coefficients */
     static float *coef_imag = 0;

     float C_real, C_imag;    /* temporary butterfly */
     float D_real, D_imag;    /* output data values */
     float *temp;

     if (N != last_N) {
          half_N = N / 2;
          if ((int)coef_real != 0) free(coef_real);
          if ((int)coef_imag != 0) free(coef_imag);
          coef_real = (float *)calloc(half_N, sizeof(float));
          coef_imag = (float *)calloc(half_N, sizeof(float));
          for (cix = 0; cix < half_N; cix++) {
               coef_real[cix] = cos(TWOPI * (double)cix / (double)N);
```

```
            coef_imag[cix] = -sin(TWOPI * (double)cix / (double)N);
        }

        num_stages = 0;
num_bits = 0;
        stage = N;          /* temporary use of this index */

        while (stage >= 2) {
            num_stages++;
            stage /= 2;
     num_bits++;
        }

        last_N = N;
if ((int)bit_reversed_index != 0) free(bit_reversed_index);
bit_reversed_index = (int *)calloc(N, sizeof(int));

for (dixA = 0; dixA < N; dixA++) {
    bit_reversed_index[dixA] = bit_reverse(num_bits, dixA);
}

   }

   skip_len = half_N;
   cix_incr = 1;

   for (stage = 0; stage < num_stages; stage++) {
        dixA = 0;
        dixB = skip_len;
        skip_pt = skip_len - 1;
        cix = 0;

        for (bix = 0; bix < half_N; bix++) {
            C_real = data_real[dixA] + data_real[dixB];
            C_imag = data_imag[dixA] + data_imag[dixB];
            D_real = data_real[dixA] - data_real[dixB];
            D_imag = data_imag[dixA] - data_imag[dixB];
            data_real[dixA] = C_real;
            data_imag[dixA] = C_imag;
            data_real[dixB] = D_real * coef_real[cix] -
                              D_imag * coef_imag[cix];

            data_imag[dixB] = D_real * coef_imag[cix] +
                              D_imag * coef_real[cix];
```

```
                if (dixA < skip_pt) {
                     dixA++; dixB++;
                } else {
                     skip_pt += 2 * skip_len;
                     dixA += skip_len + 1;
                     dixB += skip_len + 1;
                }

                cix = (cix + cix_incr) % half_N;
           }

           skip_len /= 2;
           cix_incr *= 2;
      }

      temp = (float *)calloc(N, sizeof(float));
      memcpy(temp, data_real, N * sizeof(float));

      for (dixA = 0; dixA < N; dixA++) {
 data_real[bit_reversed_index[dixA]] = temp[dixA] / (float)N;
      }

      memcpy(temp, data_imag, N * sizeof(float));

      for (dixA = 0; dixA < N; dixA++) {
 data_imag[bit_reversed_index[dixA]] = temp[dixA] / (float)N;
      }

      free(temp);

}


main()
{
    I16 card, err, card_num;
    U32 count, i;
    FILE *outfile;
    int int_count=0;

    float data_real[SIZE_FFT], data_imag[SIZE_FFT], pspec[SIZE_FFT/2];
    long data_count=0;

    printf("Please input a card number: ");
    scanf(" %d", &card_num);
```

```
    if ((card=Register_Card (PCI_7200, card_num)) <0 ) {
        printf("Register_Card error=%d\n", card);
        exit(1);
    }

//Configure PCI-7200 in handshake mode
    err = DI_7200_Config(card, TRIG_HANDSHAKE, DI_NOWAITING,
            DI_TRIG_FALLING, IREQ_RISING);
    if (err!=0) {
       printf("DI_7200_Config error=%d", err);
       exit(1);
    }


    data_count=0;
    int_count=0;
    do
    {
       //read 2 K samples from FIFO
       err = DI_ContReadPort(card, 0, DoBuf, read_count,
                             CLKSRC_EXT_SampRate, SYNCH_OP);
       if (err!=0) {
          printf("DI_ContReadPort error=%d", err);
          exit(1);
       }

       for (i=0; i<SIZE_FFT; i++)
       {
          // using ADC U3
          //data_real[i] = (float) ((DoBuf[i] & 0xff000000) >> 24);
          // Using ADC U1
          data_real[i] = (float) ((DoBuf[i] & 0x0000ff00) >> 8);
          data_imag[i] = 0.0;
       }

       FFT(SIZE_FFT, data_real, data_imag);

       for (i=0; i<SIZE_FFT/2; i++)
       {
         pspec[i] = (pspec[i]*(float)int_count + data_real[i]*data_real[i]
    + data_imag[i]*data_imag[i])/((float)(int_count+1));
       }
       data_count++;
       int_count++;
```

```
        printf("%d %d \n",data_count, int_count);

// 15000 corresponds to 1 sec when sampled at 30 MHz
        if(int_count == 15000){
           printf("Writing to file ... ");
  outfile = fopen("adc.out","a");
           for (i=0; i<SIZE_FFT/2; i++)
       fprintf(outfile,"%f\n",pspec[i]);

           fclose(outfile);
  int_count = 0;
        }

   }while(!kbhit());

   DI_AsyncClear(card, &count);
   DO_AsyncClear(card, &count);
   Release_Card(card);

}
```