**Michael Barbeau, VE3EMB**

School of Computer Science, Carleton University, 1125 Colonel By Dr, Ottawa, Ontario, Canada K1S 5B6; barbeau@scs.carleton.ca

# Programming the AD7476 Analog to Digital Converter on the *Linux*/BF537 Platform

*Use Linux to get the most from a powerful, flexible analog/digital converter for software defined radio.*

The Blackfin 537 (BF537) is a single-board computer that has the capability to run the *Linux* operating system. The Analog Devices 7476[1] (AD7476) is a small peripheral board that plugs into the BF537. It converts analog data to digital data. It can perform conversion at a rate up to 1 million Samples Per Second (SPS). Each sample is represented as 12-bit value.

The *Linux*-BF537-AD7476 combination has been introduced in Ray Mack's *QEX SDR: Simplified*[2] column as a platform for experimenting software defined radio principles. See Figure 1.

In this article I'll introduce in detail AD7476 programming in the *Linux*-BF537 environment. I will describe software that runs in *Linux* user space on the BF537. In the *Linux* environment, the AD7476 is presented to the programmer as a Serial Peripheral Interface (SPI) device abstraction in the file system space. This is an interesting feature of *Linux*. Hardware peripherals are represented as file abstractions hence most of the operations defined on files apply to peripherals. This achieves a degree of interface uniformity.

Firstly, let's review the SPI. Then, we'll walk through a program that illustrates the constraints and capabilities of the AD7476 in the *Linux*-BF537 environment. Source code is provided and made available online together with additional information about the installation on the BF537 of an appropriate *Linux* kernel.[3] Please also look at the series of articles presented in the *SDR: Simplified* column. You will find a lot of useful additional information.

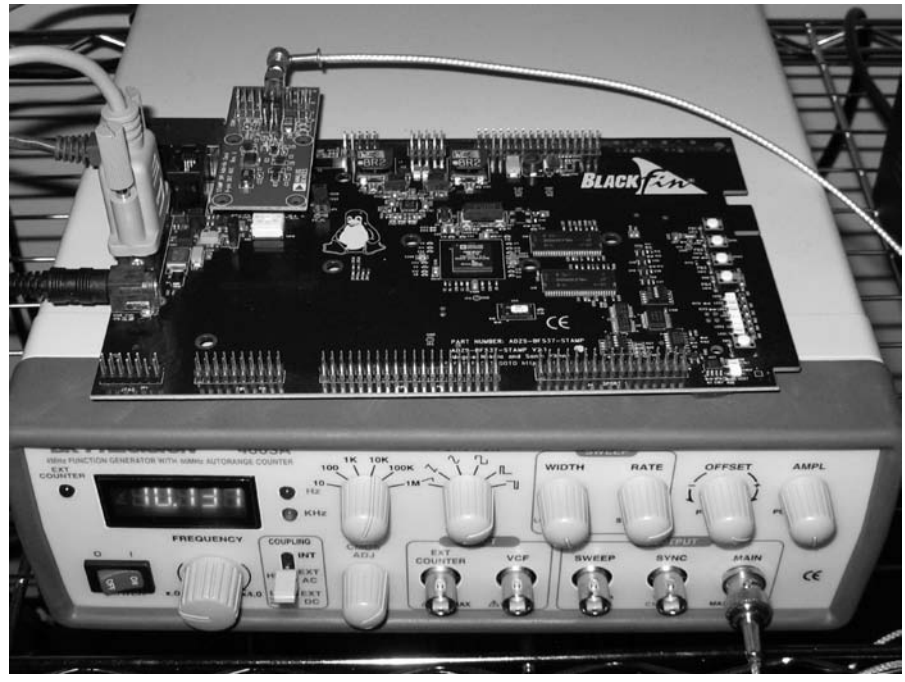


**Figure 1 -- The BF537 with the AD7476 plugged in. On the right side, a RS-232 connector linking the BF537 to a PC with a terminal emulation software proving a console to the single board computer. Using an SMB connector, the AD7476 is connected to a signal generator for testing purposes.**

## Serial Peripheral Interface

The AD7476 accepts analog input. It is connected to the BF537 according to an interface specification called the Serial Peripheral Interface (SPI). See Figure 2. The SPI defines a number of interconnection lines and the format of the signals flowing on these lines. The three key interconnection lines are the serial clock (SCK), chip select (CS) and serial data (SDATA) lines. The SCK line carries clock signals from the BF537 the AD7476. The CS line triggers the conversion of a signal from analog to digital. The SDATA line carries the digital samples from the AD7476 to the BF537.

The relationships between the SCK, CS and SDATA signals are illustrated in

[1]Notes appear on page 00.

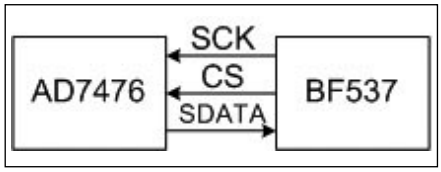


**Figure 2 -- Serial peripheral interface.**

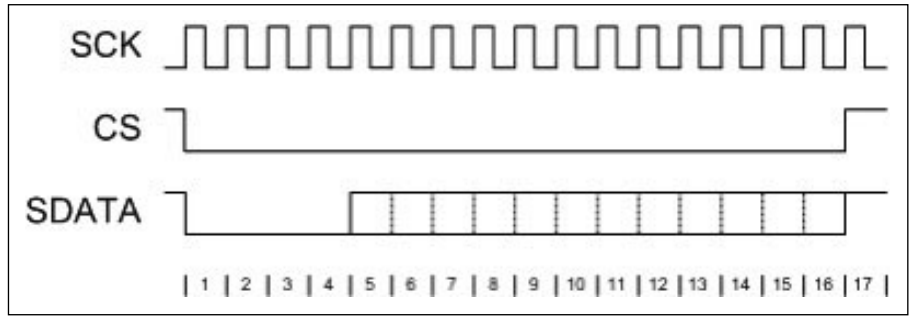


**Figure 3 -- Data flow on the SPI. The programmer has to determine the frequency of the SCK signal, which divided by 17, defines the sampling rate.**

Figure 3. The falling edge of the CS signal triggers an analog to digital conversion. The CS line is maintained low while the conversion is in progress. The AD7476 generates 12 bits of data for every sample, but 16 bits are sent. Four zero bits are sent first. Then the 12 data bits are sent, from the most significant to the least significant. The BF537 receives the bits one by one and stores them into a 16-bit word in memory. Taking into account the CS signal, each conversion requires 17 SCK cycles.

## The SPI in *Linux*

A *Linux* system with a kernel 2.6.22, or above, is needed to run a user level program that uses the SPI interface. The BF537 board had been delivered with kernel 2.6.19. You may determine the current version of the *Linux* kernel on your system by issuing the following command:

```
root:/> uname -a
```

It returns a text message similar to the following:

```
Linux blackfin 2.6.22.19-ADI-2008R1.5-svn
```

The existence of the SPI device abstraction is confirmed by entering the following command:

```
root:/> ls /dev/spi
```

that prints:

```
/dev/spi
```

You can find a 2.6.22 kernel image on my Web page. You will have to install a TFTP server on your PC for uploading the image on your board. The best option I found (no cost) so far is from Weird Solutions.[4] Don't forget to disable all firewalls on your computer (including the *Windows* firewall and any other firewall that may come with your security software). I also put on my Web page a complete kernel loading and boot example. Note that this is non destructive. The old image remains on the BF537. You can return to it, any time after a board reset.

## Sampling Rate Configuration

Let's discuss the calculation of the data rate of the output serial data stream on the SDATA line of the AD7476. First, however, an understanding of the data stream format is required.

The sampling rate of the AD7476 is a variable, denoted as *s*. According to the data sheet, the maximum sampling rate is 1 million SPS. The minimum is zero, because the AD7476 can be kept idle. The sampling rate is determined programmatically, but indirectly and according to an equation that is described hereafter. Two parameters entering in the equation are the system clock frequency and a calculated baud count. The system clock frequency, denoted as *f*, is a fixed value extracted programmatically using the system call:

```
ioctl(fd, CMD_SPI_GET_SYSTEMCLOCK, &f);
```

The function `ioctl()` can be invoked to get or set various system variables. The system clock frequency is one of the variables that can be accessed. The call to `ioctl()` is parameterized with three arguments. The first argument is a reference to the AD7476 device. The second argument is a constant that indicates the kind of access and name of system variable. The third argument is a reference to a user variable in which the result is stored when `ioctl()` returns. On the BF537, `ioctl()` returns 100 MHz. Note that the value of the system clock frequency is high compared to the sampling rate.

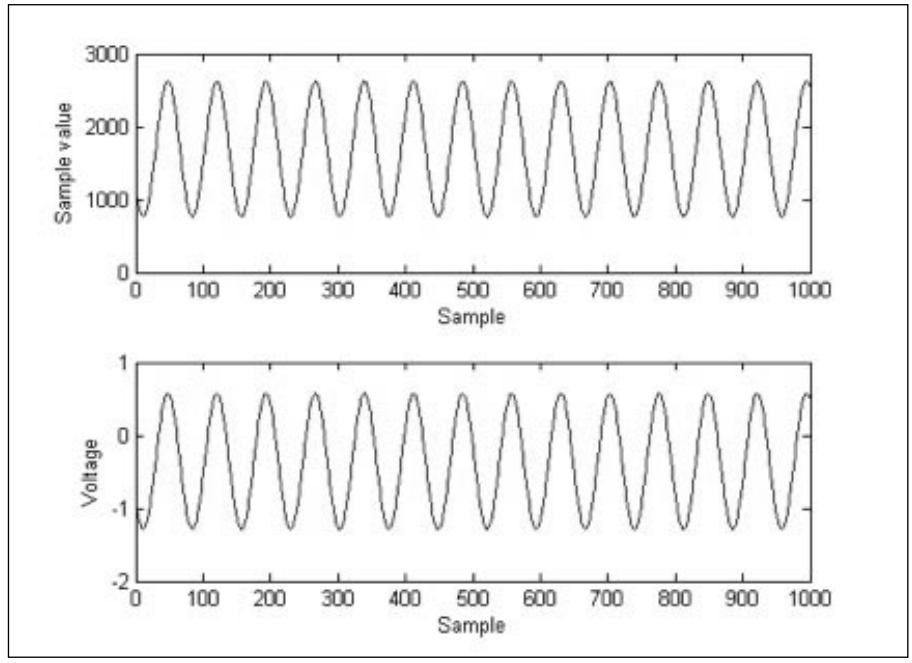


**Figure 4 -- Plotted samples in the *MATLAB* environment.**

The baud count, denoted as *b*, is a 16-bit value ranging from 1 to 65,535 bauds. The exact value is chosen such that the calculated sampling rate matches as close as possible to a desired sampling rate. First, the system clock frequency, together with the baud rate, determines the serial clock frequency, i.e., *sck*, according to the equation:

$$sck = \frac{f}{2b} \qquad \textbf{Eq. 1}$$

Every system clock cycle consists of two pulses, a low value pulse and a high value pulse. In other words, it consists of two bauds. The divisor *2b* specifies the number

of times the system clock must tick before the serial clock ticks. The serial clock ticks every *2b* bauds of the system clock. The system call `ioctl()` is used to set the serial clock frequency as follows:

```
ioctl(fd, CMD_SPI_SET_BAUDRATE, f/(2*b))
```

Each analog to digital conversion requires 17 serial clock cycles. The corresponding sampling rate, in SPS, is defined as:

$$s = \frac{sck}{17} \qquad \textbf{Eq. 2}$$

The remaining issue is the calculation of *b* such that the resulting sampling rate matches as much as possible a required sampling rate. If we substitute in Eq. 2 the symbol *sck* by Eq. 1, we obtain the following relationship:

$$s = \frac{f}{34b}$$

Since *f* is fixed, it may not be possible to pick a *b* such that the resulting sampling rate matches exactly a required sampling rate *s'*. The baud count can be picked as the smallest integer such the following relationship is satisfied:

$$s' \le \frac{f}{34b}$$

This can be rewritten as:

$$b \le \frac{f}{34s'}$$

The equation is satisfied if we pick *b* as:

$$b = \left\lfloor \frac{f}{34s'} \right\rfloor$$

The f loor operator is used. The floor of *x*, denoted as $\lfloor x \rfloor$, returns the integer part of *x*. In the C programming language, this is coded as (assuming all variables are integers):

```
b = f/(34*s);
```

*Example 1*: Let *f* be equal to 100 MHz and *s'* be equal to 1 MSPS. The expression $b = \lfloor f/34s' \rfloor$ is equal to 2. The resulting sampling rate is $s = f/34b$ or 1,470,588 SPS.

*Example 2*: Let *f* be equal to 100 MHz and *s'* be equal to 900 KSPS. The expression $b = \lfloor f/34s' \rfloor$ is equal to 3. The serial clock frequency is $sck = f/(2 \times 3) = 16{,}666{,}666$ Hz. The resulting sampling rate is $s = f/34b$ or 980,392 SPS.

According to Example 1, if we pick a *b* equal to 2, the AD7576 is over clocked and the results may not be correct. Example 2 tells us that *b* has to be at least 3 and that, by design, the maximum achievable sampling rate obtainable with the BF537-AD7476 combination is 980,392 SPS.

The core of the AD7476 board is a small IC also named AD7476. The IC has an input voltage labelled $V_{dd}$. This voltage has a double role. It powers the IC. It also serves as a reference voltage. The minimum value for a sample, zero, is produced when the level of the input analog signal is zero. The maximum value of a sample (2 power 12 minus 1 or 4095) is produced when the level of the signal matches $V_{dd}$. On the AD7476 board, this voltage is set to 4.11 V.

## Program

We discuss hereafter in detail a *C* program that reads data from the AD7476 in the *Linux* environment. The example is written according to an example provided on the Web.[5] A structure, called Data, is defined to represent information related to the AD7476.

```
typedef struct {
  unsigned short mode;
  unsigned int s;
  int fd;
  unsigned short *samples;
  unsigned int nsamples;
} Data;
```

The AD7476 can operate either in AC mode or DC mode. The mode refers to the kind of analog signal on the input line of the AD7476. The field `mode` keeps track of the programmer's selection. The required sampling rate is stored in field `s`. According to *Linux*, each hardware peripheral, e.g., hard disk or network card, is represented in the system by a device abstraction. Each such device has a name. All available devices can be viewed by listing the directory `/dev`. The AD7476 appears as a device named `/dev/spi`. Within a program, the name has to be mapped to a numerical value called a file descriptor. The field `fd` stores that reference. The programmer has to allocate areas of memory to store the samples that are received from the AD7476. The field `samples` stores a pointer to such an area. It is defined in *C* as a pointer to a buffer of unsigned short integers, i.e., 16-bit unsigned values. The length, in bytes, of that buffer is stored in the field `nsamples`. This structure is defined in a *.h* file included in the main program. We are now ready to look at the main program:

```
int main () {
  Data data;
  data.mode = AC;
  data.s = 980392;
  data.nsamples = NSAMPLES;
  AllocateMemory(&data);
  Sample(&data);
  PrintSamples(&data);
}
```

First, a variable named `data` of type `Data` is declared. In this example, the AC input mode is selected. The sampling rate is set to 980,392 SPS. The length of the buffer of samples is defined as `NSAMPLES`. The actual buffer area is allocated dynamically by the function `AllocateMemory()`. It takes an argument pointer to the variable `data`. The function allocates the space and stores a pointer to it in field `samples` of variable `data`. Next, the function `Sample()`, parameterized with a pointer to variable `data`, implements the read access to the AD7476. Finally, the function `PrintSamples()` dumps the values in a format compatible with the tools *Octave* and *MATLAB*. It is worth looking at the code of function `Sample()`.

```
void Sample(Data * data) {
  data->fd = open("/dev/spi", O_RDWR);
  setSckFreq(data->fd, data->s);
  read(data->fd, data->samples,
      data->nsamples+2) * 2);
  close (data->fd);
}
```

It reads the samples from the AD7476. For the sake of simplicity, error verification and handling are omitted. The first step consists of an invocation of the `open()` system call. The call is parameterized with the *Linux* name of the AD7476 device abstraction. It returns a numerical reference to the opened device, which is used in the following to access the device. The function `setSckFreq()` is invoked to set the serial clock frequency of the AD7476 device, first argument, and the required sampling rate, second argument. The function `setSckFreq()` implements the algorithm discussed in paragraph entitled *sampling rate configuration*. The call to the system function `read()` gets the samples from the AD7476. It is parameterized with the numerical reference to the device, a pointer to buffer area and length of buffer area in bytes. Note that space for one more sample

than the required number of samples has been allocated. For some reason, the first sample is inconsistent, read but ignored in the sequel. See Sidebar 1 for the complete code.

## Plotting the Samples

Sidebar 2 contains *MATLAB* code for plotting the sample values. The output is pictured in Figure 4. The upper part plots the values of 1000 samples. The values range from 0 to 4095. This range corresponds to a 12-bit resolution. The frequency of the input signal is 13 kHz, produced by a sine function generator. At 980,392 SPS, this corresponds to an interval of approximately one millisecond of sampling or 13.5 cycles. The file *samples.m* defines two variables, namely, samples and voltages, which are two vectors of values that must be loaded in the *MATLAB* workspace. The file *plotsamples.m* contains the plotting code.

*Michel Barbeau holds a Canadian Amateur Radio Operator's certificate and an Amateur Digital Radio Operator's certificate. He received Bachelor, Masters and PhD degrees, all in Computer Science, from the Universite de Sherbrooke in undergraduate studies and the Universite de Montreal in graduate studies. He has been a professor of Computer Science since 1991. Michel is currently working at Carleton University where he teaches and conducts research in wireless communications.*

*Michel is a member of the Ottawa Valley Mobile Radio Club, Radio Amateurs of Canada and the ARRL. His preferred modes of operation are CW, packet, PSK31, RTTY and SSB. He is also interested in amateur applications for software defined radios.*

## Notes

[1]Analog Devices, AD7476/AD7477/AD7478, **www.analog.com**, 2009.
[2]Mack, Ray, SDR: Simplified, *QEX*, Jan/Feb 2009 – Jan/Feb 2010.
[3] **http://people.scs.carleton.ca/~barbeau/SDR/**
[4] **http://corporate.weird-solutions.com/products/tftp-turbo**
[5] **http://docs.blackfin.uclinux.org/doku.php**

### Sidebar 1

### *C* Listings

File spiadc.h

```
#define CMD_SPI_SET_BAUDRATE 2
#define CMD_SPI_GET_SYSTEMCLOCK 25
#define CMD_SPI_SET_WRITECONTINUOUS 26
```

File myamradio.h

```
/*
 * Structure representing the data related to the AD7476.
 */
typedef struct {
      unsigned short mode;     // AC or DC
      unsigned int s;          // Sampling rate
      int fd;                  // Ref. to the AD7476 device
      unsigned short *samples; // Buffer of samples
      unsigned int nsamples;   // Number of samples
} Data;
```

File adc.h

```
enum { AC, DC };
#define ADC_RESOLUTION 4096      // 12-Bit (2^12)
#define REF_VOLTAGE 4.11         // Volts
#define MAX_SAMPLERATE 1000000   // 1M SPS
```

File myamradio.c

```
/* Read samples from the AD7476 card.
 */
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/poll.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

(continued)

```
#include <sys/ioctl.h>
#ifdef TM_IN_SYS_TIME
#include <sys/time.h>
#else
#include <time.h>
#endif
#include "spiadc.h"
#include "myamradio.h"
#include "adc.h"
// Number of samples
#define NSAMPLES 1000
// Debugging mode
#define DEBUG
/*
 * Allocate memory space for samples.
 */
void AllocateMemory (Data * data) {
      data->samples = malloc((data->nsamples+2) * 2);
      if (data->samples == NULL) {
            fprintf(stderr, " failed to allocate memory!\n");
            exit(1);
      }
}
/*
 * Convert sample values to voltage levels.
 */
float SampleToVoltage (unsigned short value, Data * data) {
      if (data->mode)
            // DC case
            return ((float)value/(float)ADC_RESOLUTION) * REF_VOLTAGE;
      else
            // AC case
            return (((float)value-
((float)ADC_RESOLUTION/2.0))/
(float)ADC_RESOLUTION) *
REF_VOLTAGE;
}
/*
 * Compute and set the freq. of serial clock (SCK) of the
 * SPI between the BF537 and AD7476.
 * f: reference to the AD7476 device
 * s: desired sampling rate (in SPS)
 */
void setSckFreq(int fd, unsigned int s) {
      unsigned int f;          // Sys clock freq. (in Hz)
      unsigned int b;          // Baud count
      // Get the system clock frequency
      ioctl(fd, CMD_SPI_GET_SYSTEMCLOCK, &f);
      // Compute the baud count
      b = f/(34*s);
      // Verify the sampling rate
            if (MAX_SAMPLERATE < f/(34*b)) {
```

(continued)

```
                fprintf(stderr, "Sampling rate is too high\n");
                exit(1);
        }
        // Set the serial clock frequency
        if (ioctl(fd, CMD_SPI_SET_BAUDRATE, f/(2*b)) < 0) {fprintf(stderr,
"Sck cannot be more than 33.25MHz\n");
                exit(1);
        }
}
/*
 * Read the samples.
 */
void Sample(Data * data) {
        int code;    // return code
        // Open the device representing the AD7476
        data->fd = open("/dev/spi", O_RDWR);
        if ((code=data->fd) < 0) {
                fprintf(stderr,
"Sample: failed to open /dev/spi: %d\n", code);
                exit(1);
        }
        // Set the serial clock frequency on the SPI
        setSckFreq(data->fd, data->s);
        // Read samples
        if (code=read(data->fd, data->samples,
(data->nsamples+2) * 2) < 0) {
                fprintf(stderr,
"Sample: failed to read samples: %d\n", code);
                exit(1);
        }
        // Close the device
        close (data->fd);
}
/*
 * Print the samples.
 */
void PrintSamples(Data * data) {
        int i;// loop index
// Output file descriptor
          FILE *fp;
          // open the output file: MATLAB M-file
          if ((fp = fopen("samples.m", "w")) == NULL) {
                  fprintf(stderr,
"PrintSamples: can't open output file\n");
                  exit(1);
          }
        // MATLAB array declarations
        // Print sample values (skip the first)
        fprintf(fp, "samples=[");
        for (i = 1; i < data->nsamples - 1;i++) {
                fprintf(fp, "%d, ", data->samples[i]);
        }
```

(continued)

```
            fprintf (fp, "%d];\n", data->samples[i]);
            // Print voltage values (skip the first)
            fprintf(fp, "voltages=[");
            for (i = 1; i < data->nsamples - 1;i++) {fprintf(fp, "%f, ",
      SampleToVoltage(data->samples[i], data));
            }
            fprintf(fp, "%f];\n",
      SampleToVoltage(data->samples[i], data));
            // close the output file
              fclose(fp);
      }
      /*
       * Main program.
       */
      int main () {
            // Info about the AD7476
            Data data;
            // Input mode (AC or DC)
            data.mode = AC;
            // Sampling rate
            data.s = 980392;
            // Initialization of variables
            data.nsamples = NSAMPLES;
            // Allocate the memory
            AllocateMemory(&data);
            // Sampling
            Sample(&data);
            // Print the samples
            PrintSamples(&data);
            exit(0);
}
```

## Sidebar 2

## *MATLAB* Listing

File samples.m

```
samples=[1099, 1044, 990,…];
voltages=[-0.952244, -1.007432, -1.061616,…;
```

File plotsamples.m

```
% Plot sample and voltage values
%%% Samples
subplot(2,1,1);
plot(samples);
xlabel('Sample');
ylabel('Sample value');
%%% Voltages
subplot(2,1,2);
plot(voltages);
xlabel('Sample');
ylabel('Voltage');
```

QEX