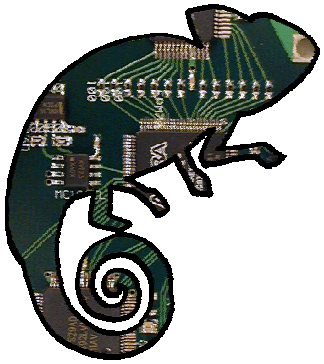

Chameleonic Radio

Technical Memo No. 18

Blackfin-Based Continuous Baseband Processing

Jang Hoon Oh and S.W. Ellingson

April 1, 2007

Bradley Dept. of Electrical & Computer Engineering
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061

Blackfin-Based Continuous Baseband Processing

Jang H. Oh
and
S.W. Ellingson

April 1, 2007

Chapter 1

Overview

Continuous baseband processing, such as FM demodulation, occurs in the Blackfin processor [2][3]. The main program for the continuous processing is implemented using C-language source code which is cross-compiled in the LINUX and executed in the μClinux operating system. All source codes are freely available at project web site [1]. In this report, we will introduce and describe the embedded program for continuous baseband processing, which includes continuous sample reception, demodulation, and sound playback in the Blackfin Board.

1.1 Main Program

Fig. 1.1 shows the flow chart of the developed main program, with which Blackfin controls all the functionality of the software radio to perform continuous baseband processing. For the software baseband processing, multiple functions are implemented in the program, including PPI interface, FM demodulation, sample decimation, and soundcard interface. Moreover, there are a couple of software techniques adopted to improve the efficiency of processing, such as circular buffering and multithreading.

The functional flow of the program can be described as the following. After starting with the initialization of required variables and buffers, it opens and configures AD 1836A soundcard and PPI port with IOCTL commands. In detail, PPI port is set to slave-mode and uses 2-dimensional DMA to transfer data to Blackfin processor. To get same amount of data from Stratix FPGA board, the receiving buffer was set to get 2048 samples at each transmission.

After reading 2048 samples in, the samples are being stored into circular buffer, which is shared by two threads, one to write in from PPI and the other to read out by soundcard. The circular buffer provides a common counter to both threads so that the two threads can trace the data amount in the buffer. To prevent the both threads from a simultaneous access to the common counter, mutual exclusion technique is in its position. Since, as mentioned previously, the samples streaming in from FPGA forms serially alternating sample format as 'IQIQIQ....' and the sample rate of 468.75 kHz is around 10 times as much as that of the maximum replay frequency of AD 1836A daughter board, these 'I' and 'Q' samples are need to be separated and downsampled for the FM demodulation and soundcard replay respectively. These manipulations of received data happens at the same time by selecting

every 40th for I channel and 41st for Q channel from the sample stream and storing the chosen sample in a pre-assigned buffer. It will reduce the original sample rate by the factor of 20 and the reduced samples will be forwarded to FM demodulator followed by a soundcard output block, which sets AD1836A daughter board with a tuned-up replay frequency for a quality playback. The program performs the sample reception and playback synchronously and continuously, if there is no user intervention to stop execution.

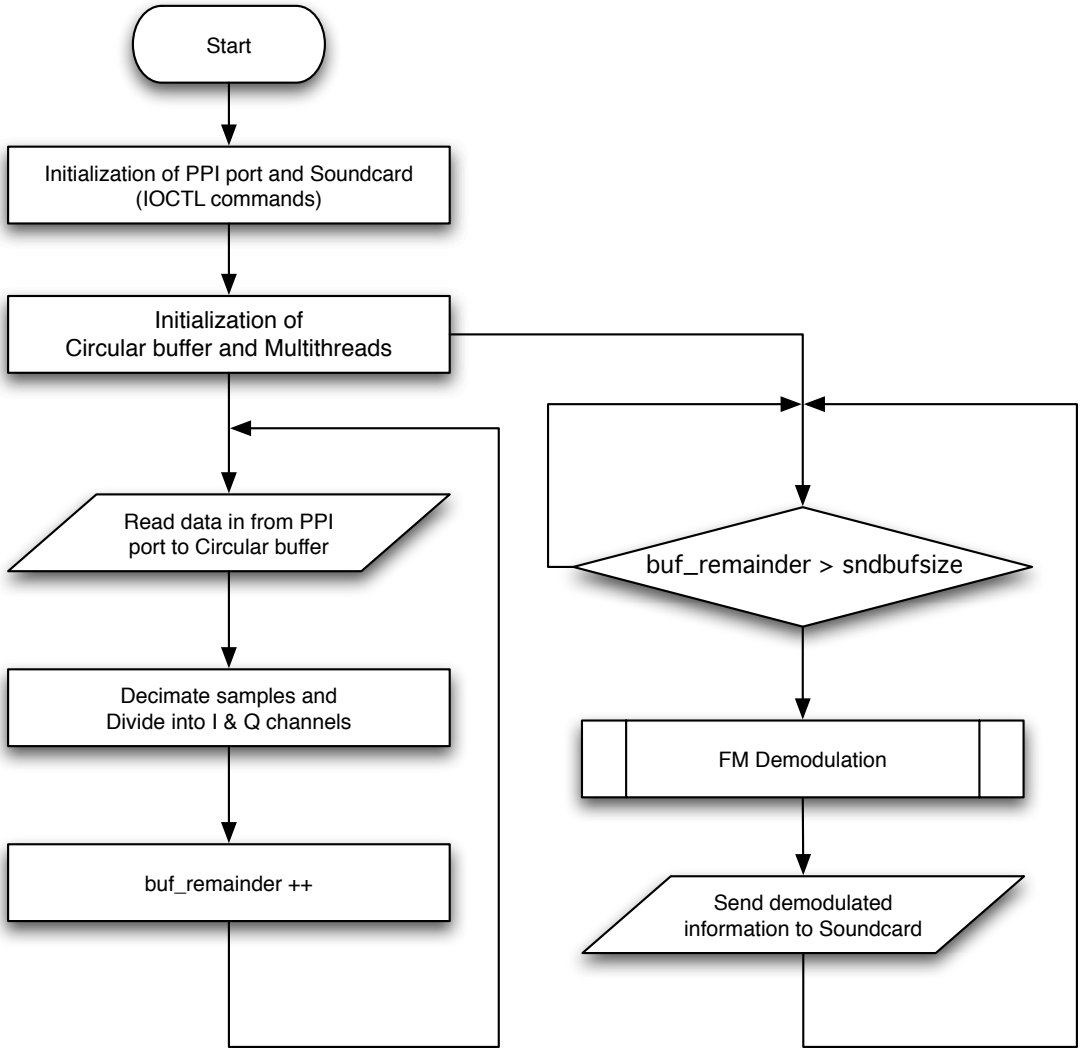

Figure 1.1: Flow chart of the main program for continuous baseband processing in blackfin.

1.2 Circular Buffering and Multithreading

In order to have the program do the continuous baseband processing, two popular software techniques, circular buffering and multithreading, are chosen and implemented. For a circular buffering, also known as ring buffering, is one of the essential features for synchronization between data input and output. While its name is figurative, the circle intimates

that the read/write position in the buffer will rotate to accommodate a continuous data I/O. It is crucial that a reader and a writer of a circular buffer share a position indicator so that they can keep the pace with each other. If a reader faster than a writer, a buffer will get underflowed. On the other hand, with faster writer, a buffer will get overflowed.

Therefore, to prevent these abnormal situations and make sure the synchronization between a reader and writer, a shared indicator is essential in the systems having asymmetric I/O speeds. Moreover, since the soundcard playback is done by sample blocks in the continuous baseband program, a position indicator of a circular buffer should be a combined version of a sample-wise and a block-wise positions. To meet these purposes, the variable `buf_remainder` works as the sample-wise and block-wise indicator for the synchronization. By tracking the indicator, PPI input and soundcard output blocks can let each other be aware of the position in the buffer by modifying the `buf_remainder` directly.

Since a traditional 'single-threaded' operation will perform buffer read and write activities in a time-sequence manner, if the reading and writing rates are not balanced properly or one of processes spends some time to wait for a linked routine's execution, it will degrade and curb the performance of the buffer access to some extent. For example, even not to loose any information being read in from PPI interface, the program needs to finish the FM and soundcard processing before the start of new data input from PPI. This will remarkably restrain the program from being flexible. To mitigate the restrictions on performance, the multithreading technique has been applied.

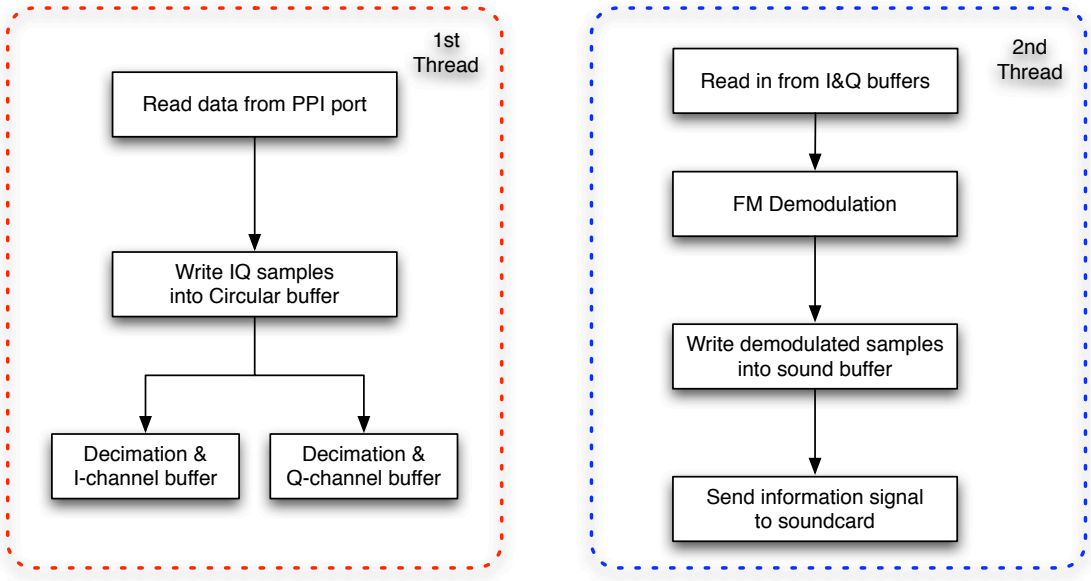

Figure 1.2: Flow diagram of the multi threading in blackfin code.

Fig. 1.2 shows the flow diagram of the multithreading in blackfin. We have used two threads - one of them reads data from the PPI port and stores them into a circular buffer and the other one do all the baseband processing and sends it to the sound card for the information playback. Since, in the multithreaded program, these two threads are running concurrently, we are not loosing any data from the PPI port and can receive it seamlessly. The multithreading is able to activate multiple threads and let them be executed in parallel

and independently within the context of single process, sharing the state information, memory and other resources directly. By taking advantage of the parallel execution, it is not needed to arrange the tasks in a time-sequence and it is possible to implement a program which is able to do continuous PPI data reception and soundcard playback without having a burden of the time-sequential alignment.

Because the multiple threads shares program resources, such as global variables and address spaces within one process, when we use more than one thread changing the common resources, it is essential to ensure the data integrity of the shared resources among threads[4]. As mentioned above, in this continuous baseband processing, it is inevitable that all the threads are using a common circular buffer and indicator to transfer data from PPI input to soundcard output. Even though all the threads synchronized by sharing the same buffer indicator, it is not able to completely prevent threads from overlapping and cancelling out the modification by other threads on shared resources. For example, if the second thread tries to read from or overwrite a certain variable, while one thread is modifying it, the data in the variable will be inconsistent and unpredictable.

Therefore, even though multithreading technique is being used, letting the shared variables accessed exclusively is very important to keep the data consistency in the so-called "critical region" of multithreaded programs. For the exclusive access to such shared resources, mutual exclusion (often called `mutex`) algorithm has been adopted. The `mutex` algorithm is generally used to avoid simultaneous access to shared resources, such as a global variable, in multithreaded programs. The mutual exclusion object can be enabled or disabled by locking and unlocking respectively of a target variable as in the following sample code:

Mutex Code

```
#include <pthread.h>

        ...

pthread_mutex_t mLock=PTHREAD_MUTEX_INITIALIZER;

        ...

pthread_mutex_lock(&mLock);
buf_remainder++;
pthread_mutex_unlock(&mLock);
```

The calls, `pthread_mutex_lock()` and `pthread_mutex_unlock()`, perform a crucial function in multithreaded programs. They literally offer a means of mutual exclusion by locking and unlocking. By using the calls, no two threads can access the same mutex locked common resource simultaneously[4]. Pthreads in the sample, are the libraries for POSIX threads standard, which are most commonly used on POSIX systems, such as LINUX and UNIX.

1.3 PPI Port Configuration and Complementary Enhancements

All of the PPI configuration including the hardware setup, cable connection and driver installation has been described comprehensively in [2]. Table 1.1 just shows the input output control (IOCTL) which have been used to configure the PPI port for running this continuous baseband processing program.

Table 1.1: IOCTL commands for configuring the PPI registers [2].

IOCTL Command	**Settings**	**Description**
CMD_PPI_PORT_DIRECTION	CFG_PPI_PORT_DIR_RX	Set to receive data
CMD_PPI_XFR_TYPE	CFG_PPI_XFR_TYPE_NON646	Set to non 646 mode
CMD_PPI_PORT_CFG	CFG_PPI_PORT_CFG_XSYNC23	Data is controlled by two external control signals (FS-1 and FS-2)
CMD_PPI_FIELD_SELECT	CFG_PPI_FIELD_SELECT_XT	Select the external trigger
CMD_PPI_PACKING	CFG_PPI_PACK_DISABLE	Data packing is disabled
CMD_PPI_SKIPPING	CFG_PPI_SKIP_DISABLE	Data skipping is disabled
CMD_PPI_DATALEN	CFG_PPI_DATALEN_16	Set data length to 16 bit
CMD_PPI_CLK_EDGE	CFG_PPI_CLK_EDGE_RISE	FS-1 and FS-2 treated as rising edge
CMD_PPI_TRIG_EDGE	CFG_PPI_TRIG_EDGE_RISE	PPI samples data on rising edge of the clock
CMD_PPI_SET_DIMS	CFG_PPI_DIMS_2D	Two dimensional data transfer
CMD_PPI_DELAY	0	There is no delay between the control signals and the starting of the data transfer
CMD_PPI_SETGPIO	-	Set the PPI to general purpose mode

Since there were glitches in the transmitted data from the Stratix board with the original configuration, two measures were taken. One is ground enhancement and the other is a new interconnection cable which has shorter length and more ground pins. The changes can be found in Fig. 1.3. After adopting the enhanced interface cable, the glitches disappeared and the continuous data capturing from PPI was actually done without errors.

1.4 FM Demodulator with Predefined NCO arrays

Fig. 1.4 shows how the FM demodulator has been implemented into the Blackfin board. This demodulator function is based on the PLL and NCO. In the continuous baseband processing program, the FM demodulator and soundcard driving block were implemented in the `soundproc()` function. In the function, decimated I and Q channel samples are transfered to get temporal phase error by subtraction between I samples and Q samples, multiplied by `sine` NCO and `cosine` NCO respectively. This resulting phase error will be forwarded to scaling and gain blocks, which should be set carefully to achieve properly demodulated information signal. After these scaling and gain blocks, the resulting signal is

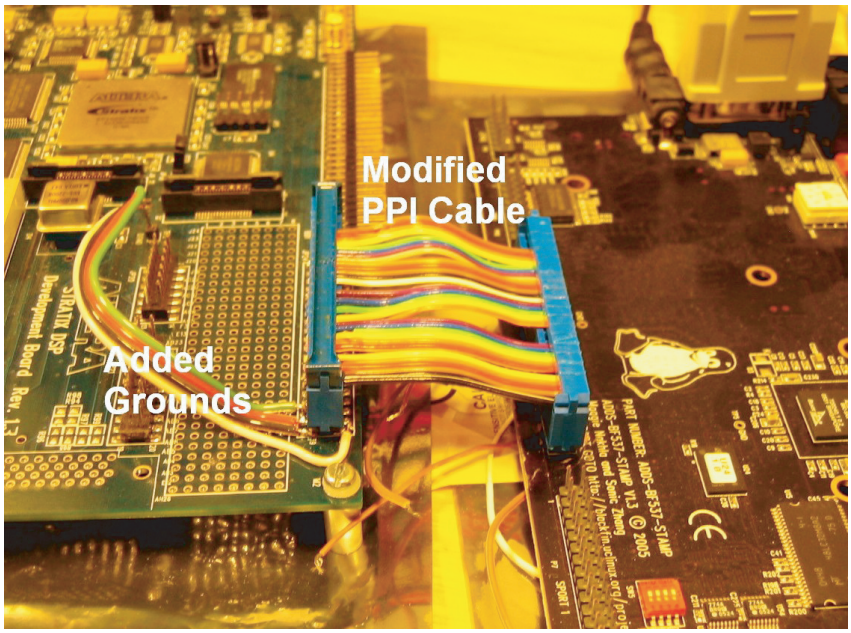

Figure 1.3: Picture of newly added ground cables and modified PPI cable

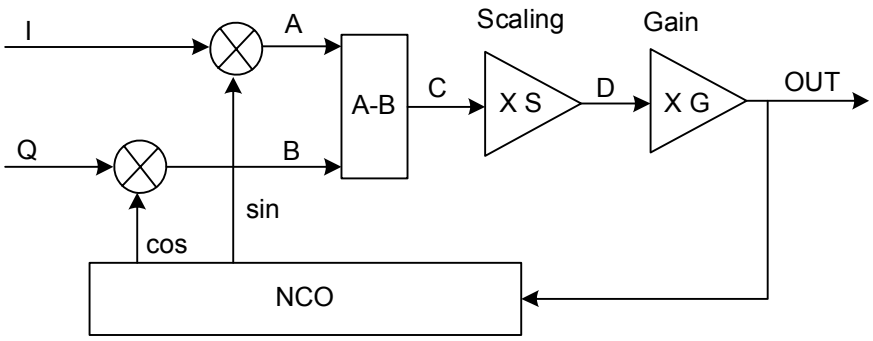

Figure 1.4: Implementation of FM demodulator.

the demodulated information samples to drive soundcard. These also be fed back to NCO arrays to adjust NCO functions.

The `soundproc()` function is initialized as a second thread in the main program to enable data capturing from PPI and soundcard playback are executed concurrently for the continuous baseband processing. Even though the multithreading is being used in the program, in order to let the program replay the data continuously, it is necessary that two threads are working synchronously without affecting any time-lag to each other.

When the built-in `soundproc()` function, which contains FM demodulator and soundcard driving block, plays its role in the main program, it took somewhat longer time in the soundcard driving block than expected. The delay in the function influenced on the overall performance of continuous FM demodulation and sound playback blocks, resulting in a discontinuous sound playback. To reduce processing time in the `soundproc()` function, the NCOs are replaced with a predefined arrays as in the following C-language code block.

NCO Tables code

```
for(i=0;i<628;i++) {
        cos_tab[i] = (short)32000*cos(twopi*(float)i/628.);
}

for(i=0;i<628;i++) {
        sin_tab[i] = (short)32000*sin(twopi*(float)i/628.);
}

        ...

if (freq_index_tmp > 627) {
        freq_index_tmp -= 627;
}
nco_i_in = cos_tab[freq_index_tmp];
nco_q_in = sin_tab[freq_index_tmp];
```

Chapter 2

Demonstration

In this section we describe a demonstration of the radio described in the previous sections. Fig. 2.1 shows the setup of the demonstration [3]. The following devices are involved:

1. A Stanford Research Systems DS345 arbitrary function generator generates the FM modulated-signal. This instrument is limited to carrier frequencies of 30 MHz or less, so a center frequency of 25 MHz was used. For the purposes of this demonstration, narrowband FM signals with information rate from 100 Hz to 4 kHz and deviation of 4 kHz were used.

2. The Altera DSP-BOARD/S25 evaluation board, which includes the A/D and the Stratix EP1S25 FPGA.

3. The Analog Devices ADDS-BF-537 STAMP Blackfin evaluation board, which is interfaced to the FPGA board as described in [2].

4. The Analog Devices AD1836A-DBRD audio daughterboard, which includes a codec and provides audio input and output. The integration of this board with the Blackfin board is documented in [6]. To avoid conflicts with the PPI, which shares pins with SPORT1, the interface between the Blackfin and the audio codec is implemented on SPORT0.

5. A laptop PC. The ethernet connection is used in the initial boot of the Blackfin processor, and the serial connection is used to program the Blackfin board (including download of the desired μClinux kernel and the application) and to establish a command line interface into the μClinux OS running on the Blackfin. The program Kermit is used for the serial connection and the program tftpboot is used to download the kernel.

6. A spectrum analyzer is connected to the D/A output of the FPGA board for testing of the transmit path. (Although not employed in the demonstration described here.)

This demonstration has been attempted, and in fact when the system is running, one can hear a continuous audio playback ranging from 0.1 to 4 kHz along with the frequency change at function generator. For further verification, we captured over 30000 samples of data from both the input and output of the FM demodulator under a fixed frequency input. The portion of resulting graph is shown in Fig. 2.2.

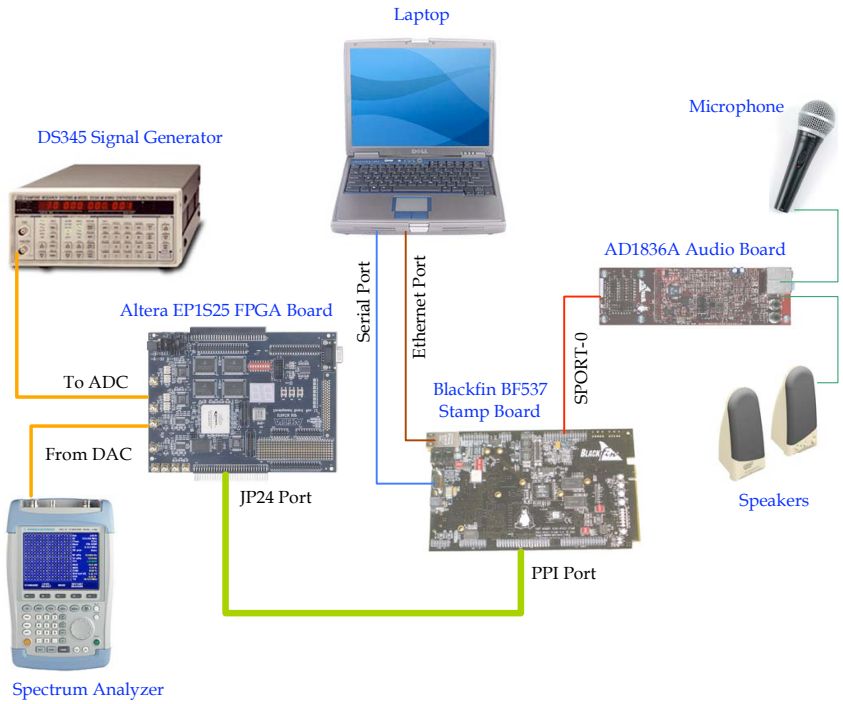

Figure 2.1: Demonstration setup.

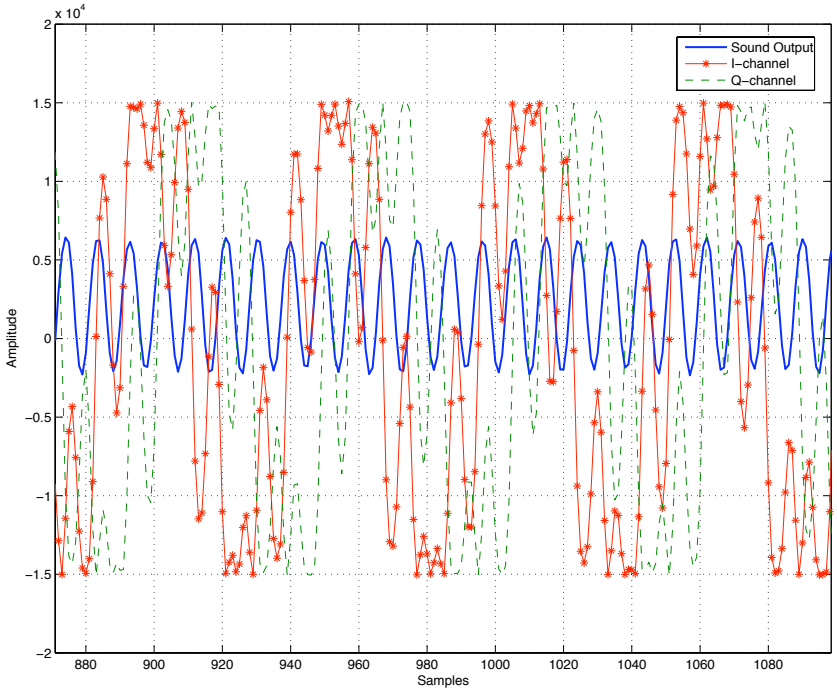

Figure 2.2: Data captured from Input and Output of FM demodulator.

Bibliography

[1] *'Virginia Tech Project Web Site'*, http://www.ece.vt.edu/swe/chamrad.

[2] S.M. Hasan and Kyehun Lee, *Interfacing a Stratix FPGA to a Blackfin Parallel Peripheral Interface (PPI)*, Technical Memo No. 7, July 23, 2006, Available on-line: http://www.ece.vt.edu/swe/chamrad/.

[3] S.M. Hasan, Kyehun Lee, and S.W. Ellingson, *FM Waveform Implementation Using an FPGA-Based Enbdded Processor*, Technical Memo No. 7, July 23, 2006, Available on-line: http://www.ece.vt.edu/swe/chamrad/.

[4] Daniel Robbins, *Common threads: POSIX threads explained, Part 2*, http://www-128.ibm.com/developerworks/library/l-posix2/.

[5] J. G. Proakis, M. Salehi, *Communication Systems Engineering*, 2nd Edition, Pearson Education, ISBN 81-7808-610-7.

[6] *AD1836A Audio Codec Card*, `http://docs.blackfin.uclinux.org`, September 20, 2006.

[7] *Visual DSP++ 4.0: C/C++ Compiler and Library Manual for Blackfin Processors*, Revision 3.0, January 2005, http://www.analog.com.