

Baseband Orthogonal Frequency Division Modem

EECS 452: Final Project Report

Awais Kamboh, Krispian Lawrence, Saradwata Sarkar, Eric Williams

Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor

19 December 2005

Abstract

A problem plaguing typical digital communication systems is multipath propagation. At radio frequencies, this can be caused by buildings, walls, and metal objects. Orthogonal Frequency Division Multiplexing (OFDM) is a method of using multiple carriers to both reduce inter-symbol interference and combat the effects of multipath propagation.

We have implemented a compile-time configurable OFDM modem on a Texas Instruments TMS320 C5510 Digital Signal Processing Starter Kit using 16-point quadrature amplitude modulation (16-QAM). This report presents a background on OFDM, the methods we used to accomplish full-duplex operation, evaluates the performance of the modem, and suggests further improvements.

Contents

| | |
|--|-----------|
| List of Figures | iii |
| List of Tables | iii |
| Acknowledgements | iv |
| Executive Summary | v |
| 1 Introduction | 1 |
| 2 Orthogonal Frequency Division Multiplexing | 1 |
| 2.1 Theory | 1 |
| 2.1.1 Multipath Propagation | 2 |
| 2.1.2 Noise | 2 |
| 2.2 OFDM Generation | 2 |
| 2.2.1 Buffering and Block Processing | 3 |
| 2.2.2 Quadrature Amplitude Modulation (QAM) | 3 |
| 2.2.3 Inverse Discrete Fourier Transform (IDFT) and Discrete Fourier Transform (DFT) | 3 |
| 2.2.4 Guard Interval | 4 |
| 3 System Design | 5 |
| 3.1 Synchronization | 5 |
| 3.2 Channel Estimation | 6 |
| 3.3 Header Frame | 7 |
| 4 Implementation | 8 |
| 4.1 Packet Structure | 8 |
| 4.2 Implemented Modules | 8 |
| 4.2.1 UART Module | 9 |
| 4.2.2 OFDM Module | 10 |
| 4.2.3 IF / CODEC Module | 11 |
| 4.2.4 CORR Module | 11 |
| 4.3 Transmitter Implementation | 11 |
| 4.4 Receiver Implementation | 11 |
| 4.5 Hardware Design Issues | 12 |
| 4.6 DSP Implementation issues | 13 |
| 4.6.1 Math Processing | 13 |
| 4.6.2 IF Module | 13 |
| 4.6.3 UART Module | 14 |
| 4.6.4 Optimizations | 15 |
| 5 Results and Findings | 15 |
| 5.1 Improvements | 15 |
| 6 Commercial Markets and Uses | 16 |
| 7 Conclusion | 18 |
| References | 18 |
| Glossary | 18 |

| | |
|--------------------|------------|
| Appendix A | A-1 |
| ofdm.c | A-1 |
| if.asm | A-11 |
| uart.c | A-16 |
| corr.asm | A-20 |
| ofdm.cmd | A-24 |

List of Figures

| | | |
|----|---|----|
| 1 | Bandwidth Allocation | 1 |
| 2 | System Schematic | 2 |
| 3 | Sample Constellation | 3 |
| 4 | OFDM Generation Block Diagram | 5 |
| 5 | Burst Frame Signal | 5 |
| 6 | Burst Frame Autocorrelation | 6 |
| 7 | Channel Estimation Constellation | 7 |
| 8 | Inverse Channel Transfer Function | 8 |
| 9 | Header Constellation | 9 |
| 10 | Data Packet Form | 9 |
| 11 | IFFT Bin Packing | 10 |
| 12 | Transmission Data Flow | 11 |
| 13 | Transmission State Machine | 12 |
| 14 | Reception Data Flow | 13 |
| 15 | Reception State Machine | 14 |
| 16 | Circular Buffer Linearization | 16 |
| 17 | BER Prediction | 17 |

List of Tables

| | | |
|---|---------------------------------------|----|
| 1 | Sample Constellation Points | 3 |
| 2 | Packet Length | 10 |
| 3 | IF Module Functions | 14 |

Acknowledgements

We would like to thank our professor, Dr. Kurt Metzger, for his support, advice and constructive comments; and Jason Siegel and Niloufar Ghafouri for their constant support throughout the course.

Executive Summary

Typical digital communication systems are prone to errors caused by multipath propagation (additive echoes). The effect of such echoes is to cause time spreading of the symbols, which prevents effective decoding. Orthogonal Frequency Division Multiplexing is a novel modulation technique that combats these traditional problems at the expense of additional algorithmic complexity.

OFDM signals can be generated with the use of typical Fast Fourier Transform and Inverse Fast Fourier Transform functions available on most digital signal processing chips. By using these, some creative input and output buffering, and a proper detection method, it is relatively easy to implement such a system.

We have done just this, using a Spectrum Digital C5510 Digital Signal Processing Starter Kit. The internal program is structured to be able to run while both sending and receiving data. Full-duplex operation is important for reliable and predictable functioning.

Our Orthogonal Frequency Division Multiplexing system has a widely compatible serial data port. A computer or similar device can communicate, without error correction, over a link to another such equipped Starter Kit. Using a number of optimized methods, we have gained good performance for a proof-of-concept project.

1 Introduction

Since the inception of modern communication theory, most communication systems have taken a single-carrier approach, where all the information to be transmitted is modulated by a single carrier. A single-carrier system uses the entire bandwidth available for each symbol, causing the data symbols to have a short time duration. Inter-Symbol Interference (ISI) can affect each symbol significantly. In a classic communication system, ISI causes severe degradation of the system performance.

Orthogonal Frequency Division Multiplexing (OFDM) is a modulation method for communication using multiple carriers spaced correctly and evenly in the frequency domain. Since OFDM allows adjacent carrier frequencies to be very closely spaced, more closely than most other multi-carrier systems, systems using this modulation scheme can use the bandwidth efficiently. Also, these systems are largely immune to the effects of multipath when correctly implemented.

We have implemented an Orthogonal Frequency Division Multiplexing modulation-based communications link. It is fully bi-directional, has a RS-232 serial interface, and it meets the behavior requirements of a Data Communications Equipment device. This work was done on a Spectrum Digital C5510 Digital Signal Processing Starter Kit. We have achieved a functioning system with promising results.

2 Orthogonal Frequency Division Multiplexing

2.1 Theory

In today's world, with the ever increasing demand for faster, secure and more reliable communication systems, multi-carrier systems are an alternative and effective approach. In a multi-carrier system, the available bandwidth is split into several sub-channels (Figure 1).

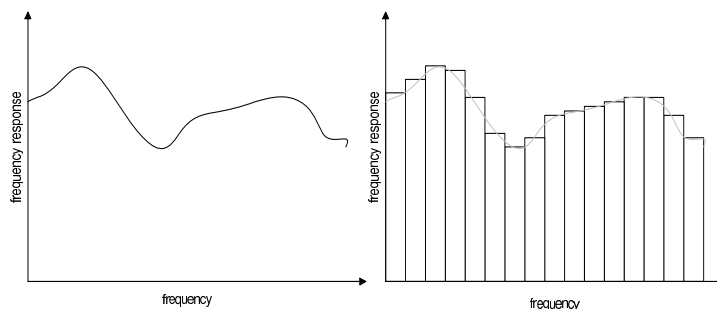


Figure 1: Splitting of bandwidth among different carriers in a multi-carrier system.

OFDM is a multi-carrier system where data is encoded to multiple sub-carriers, which are sent simultaneously. This results in an optimal use of bandwidth. A set of orthogonal sub-carriers together forms an OFDM symbol. To avoid ISI due to multipath propagation, successive OFDM symbols are separated by a guard interval. This makes the OFDM system resistant to multipath effects.

In a multi-carrier system, instead of transmitting information all at once, it is transmitted slowly in parallel over these sub-channels. This enables data symbols to have a longer duration while still maintaining high data rates. In the frequency domain, each sub-channel occupies a small frequency interval where the channel frequency response will be almost constant; each symbol will hence experience an approximately flat-fading channel.

OFDM is a specialized form of multi-carrier communication where the sub-carriers are orthogonal to one another. By using orthogonal sub-carriers, the Inter-Carrier Interference (ICI) will be nearly eliminated in practice, and the symbols transmitted on the different sub-channels will not interfere.

2.1.1 Multipath Propagation

When traveling in an analog channel, such as electromagnetic waves along a wire, or sound waves in a medium, signals frequently get compounded with delayed, distorted versions of themselves. An echo is a classic example of multipath sound propagation. In a digital communication system, multipath propagation causes frequency shaping. To effectively communicate over a channel where this happens, the modem must either know or be able to estimate the frequency-shaping effects. The technique developed for use in OFDM systems is to send a channel estimation, or pilot, signal, which is known to both the sender and the receiver. By comparing what is received against what is expected to be received, the channel may be estimated to any level of detail desired.

2.1.2 Noise

The channel adds Additive White Gaussian Noise (AWGN) to the OFDM signal waveform. AWGN is a zero-mean wide-sense stationary random process consisting of independent and identically distributed Gaussian random variables. Noise of this type has infinite power and variance. This noise model is appropriate for our modeling situation.

2.2 OFDM Generation

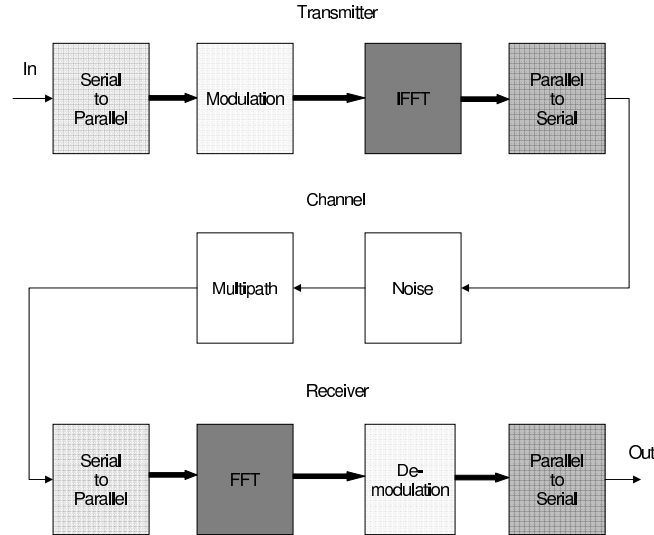


Figure 2: System schematic of the implemented modem.

To generate OFDM successfully, the relationship between all the carriers must be carefully controlled to maintain the orthogonality of the carriers. For this reason, OFDM is generated by first building the desired spectrum, based on the modulated input data. Each carrier is assigned some data to transmit. The required amplitude and phase of the carrier is then calculated based on the QAM modulation scheme. The required

spectrum is then converted to its time domain representation using an Inverse Fast Fourier Transform. A Fast Fourier Transform, at the receiver end, does the reverse during demodulation.

Consider the block diagram of a sample OFDM system shown in Figure 2.

2.2.1 Buffering and Block Processing

The input serial data stream is formatted into blocks of the size required for transmission. The data is then transmitted in parallel by assigning each data word to one carrier in the transmission. There must be enough data available before this process starts, because the data is consumed in blocks, not a byte at a time. Similarly, the data from the IFFT must be transformed from a block into a serial set of data.

2.2.2 Quadrature Amplitude Modulation (QAM)

To increase the data rate, the 16-point quadrature amplitude modulation scheme (16-QAM) is used on each sub-carrier. 16-QAM maps four bits onto one complex-valued symbol. Gray coding is also used, making adjacent symbols differ by only one bit. This makes it optimal for a minimum Euclidean distance receiver. A sample gray-coded 16-QAM constellation is shown in Figure 3 and Table 1.

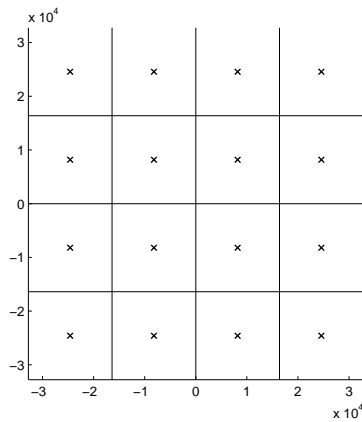


Figure 3: Sample 16-QAM constellation.

| | | | |
|----------|----------|-----------|-----------|
| 0000 (0) | 0001 (1) | 1001 (9) | 1000 (8) |
| 0010 (2) | 0011 (3) | 1011 (11) | 1010 (10) |
| 0110 (6) | 0111 (7) | 1111 (15) | 1110 (14) |
| 0100 (4) | 0101 (5) | 1101 (13) | 1100 (12) |

Table 1: A set of gray-coded 16-QAM constellation points.

2.2.3 Inverse Discrete Fourier Transform (IDFT) and Discrete Fourier Transform (DFT)

The IDFT performs the necessary transformation very efficiently; it provides a simple way of ensuring the carrier signals produced are orthogonal.

Consider the N complex-valued symbols $X(k)$, $0 \leq k \leq N - 1$, modulated onto N orthogonal carriers using the IDFT

$$x(n) = \sum_{k=0}^{N-1} X(k)e^{+j2\pi n \frac{k}{N}} \quad (1)$$

Since the basis functions of the IDFT are orthogonal, orthogonal sub carriers are created.

The Discrete Fourier Transform (DFT) transforms a cyclic time domain signal into its equivalent frequency spectrum. This is done by finding the equivalent waveform, generated by a sum of orthogonal sinusoidal components. The amplitude and phase of the sinusoidal components represent the frequency spectrum of the time domain signal. The IDFT performs the reverse process, transforming a spectrum (amplitude and phase of each complex frequency component) into a time domain signal. The IDFT converts a number of complex data points into the time domain signal of the same number of points. Each data point in the frequency spectrum used for an IDFT is called a bin. The orthogonal carriers required for the OFDM signal can be easily generated by setting the amplitude and phase of each bin, then performing the IDFT.

To achieve a real-valued output from the IDFT, a special packing technique is used. By appending a reversed and complex conjugated copy of the N symbols to themselves, the output from the IDFT will be real-valued (specifically, it will contain zero-valued imaginary components). By using this technique, the number of points in the IDFT calculation will be increased from N to at least $2N + 2$. In this case, the complex frequency bins with equal but opposite frequencies then contain conjugate coefficients.

An property that is important in the reception of OFDM signals is the close relationship between the DFT of a signal and the DFT of the same signal circularly rotated. Taking the DFT of signal rotated by R samples,

$$y(n) = x((n + (N - R)) \bmod N) \quad (2)$$

$$Y(k) = e^{-j2\pi k \frac{N-R}{N}} X(k) \quad (3)$$

The coefficient is simply a linear phase shift. This follows directly from the Fourier shifting property.

2.2.4 Guard Interval

One of the most important properties of OFDM transmission is the robustness it provides against multipath delay spread. This is achieved by having a long symbol period, which minimizes the Inter-Symbol Interference (ISI). The level of robustness can be increased even more by the addition of a guard period between transmitted symbols. The guard period allows time for multipath signals from the previous symbol to die away before the information from the current symbol is gathered. If the end of the symbol waveform is put at the start of the symbol during the guard period, this effectively extends the length of the symbol, while maintaining the orthogonality and periodicity of the waveform.

A technique for employing the guard interval is to use a cyclic prefix. The cyclic prefix is, a copy of the M of the last samples prepended, making the signal appear as periodic over $M + N$ samples with period N . The received signal, consisting of the sent signal and the cyclic prefix, is demodulated using the FFT.

The sent signal can be written as:

$$s(n) = \begin{cases} x(n + N) & , -M \leq k < 0 \\ x(n) = \sum_{k=0}^{N-1} X(k)e^{+j2\pi n \frac{k}{N}} & , 0 \leq k \leq N - 1 \end{cases} \quad (4)$$

The received signal $r(n)$ can be written as

$$r(n) = s(n) * h(n) + e(n) \quad , -M \leq n \leq N - 1 \quad (5)$$

where $h(n)$ is the channel impulse response and $e(n)$ is the error due to additive noise.

3 System Design

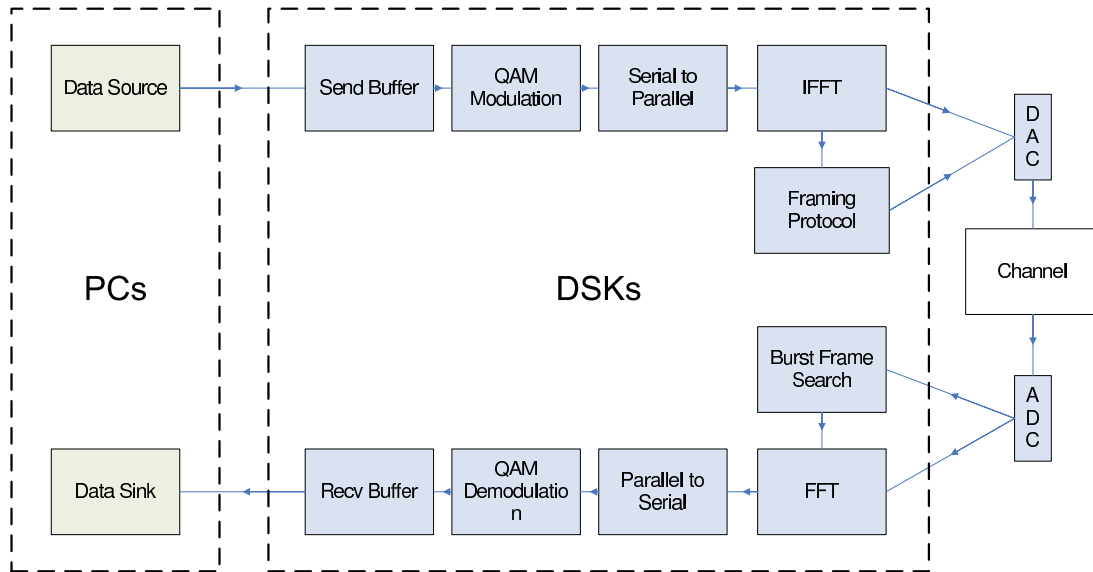


Figure 4: Block Diagram of a sample OFDM system.

A schematic of the system model that was implemented is shown in Figure 4. It consists of two PCs and two DSP-implemented modems. The same program is executed on both DSPs and duplex communication is possible. The main design issues are discussed below.

3.1 Synchronization

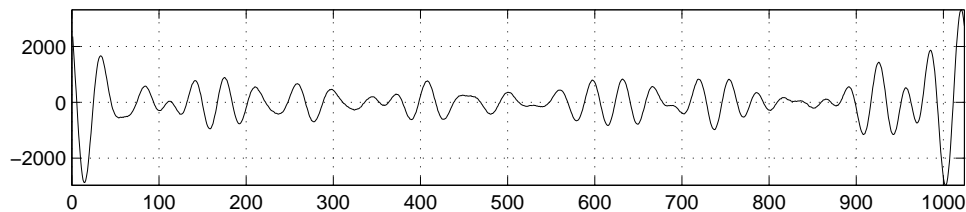


Figure 5: Time domain burst frame signal.

Proper time synchronization is an issue in any coherent communication system. To solve this problem and to reduce the complexity of the system, a pseudo-random sequence was designed to act as a Burst Frame (Figure 5). At the receiver end, a normalized running correlation is calculated between the incoming data

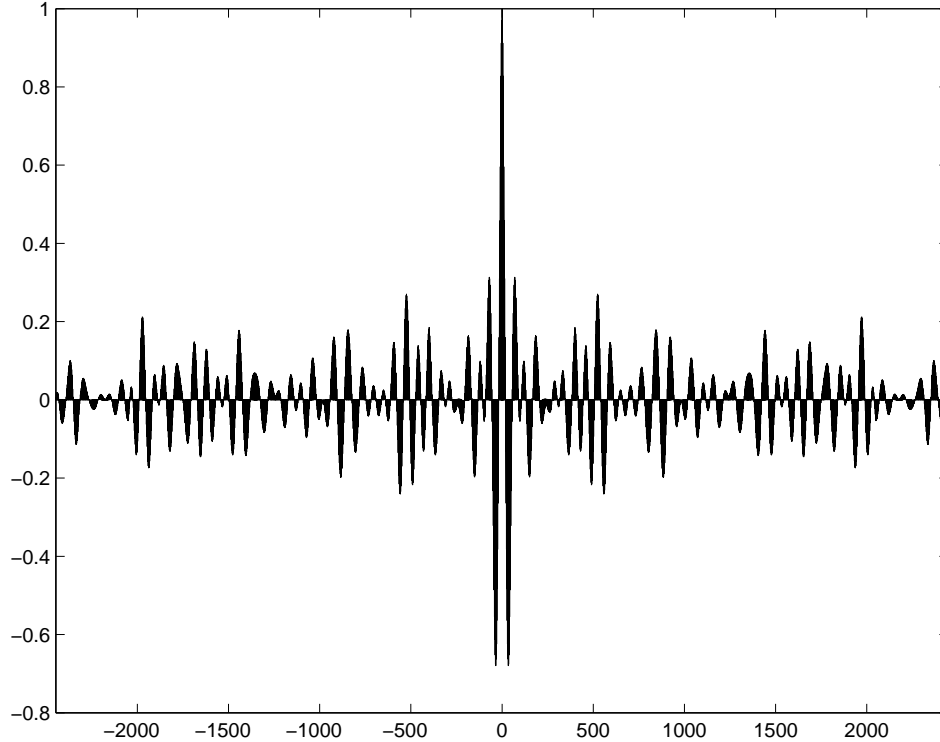


Figure 6: Autocorrelation of burst frame.

and the burst frame. A value above a pre-determined threshold indicates the arrival of a packet; the threshold is determined from the burst-frame autocorrelation properties.

The burst frame is one of the most important pieces of information sent in a packet. It is responsible for time synchronization between the transmitter and the receiver. Its optimal design is such that its autocorrelation is an impulse. The normalized autocorrelation for our chosen burst frame is shown in Figure 6. It only has a peak of substantial magnitude in the center, which is a desired characteristic and helps in correct detection at the receiver. Because of the DFT shifting property discussed earlier, and the channel estimation and correction frame discussed below, detection of any of these three (two of them negative) peaks will synchronize the transmitter and receiver to within tolerance.

3.2 Channel Estimation

Since we may deal with a time-varying channel, a continuous estimate of the channel behavior is important to ensure reliable data transmission and reception. To cope with this, a known pilot frame is sent, and the received frame (Figure 7(b)) is compared with the original frame and the channel response is estimated.

The channel estimation frame is another important frame that predicts the channel response at the receiver. It is a set of all the possible symbols assigned to each subchannel as shown in Figure 7(a). These symbols face deterioration and attenuation while traveling through the channel resulting in a new skewed constellation. Figure 7(b) shows the constellation of the received channel estimation frame.

Once the channel estimation frame is received, the information is used to generate a channel adjustment matrix. The channel adjustment matrix acts as a linear transform applied to all of the following frames to

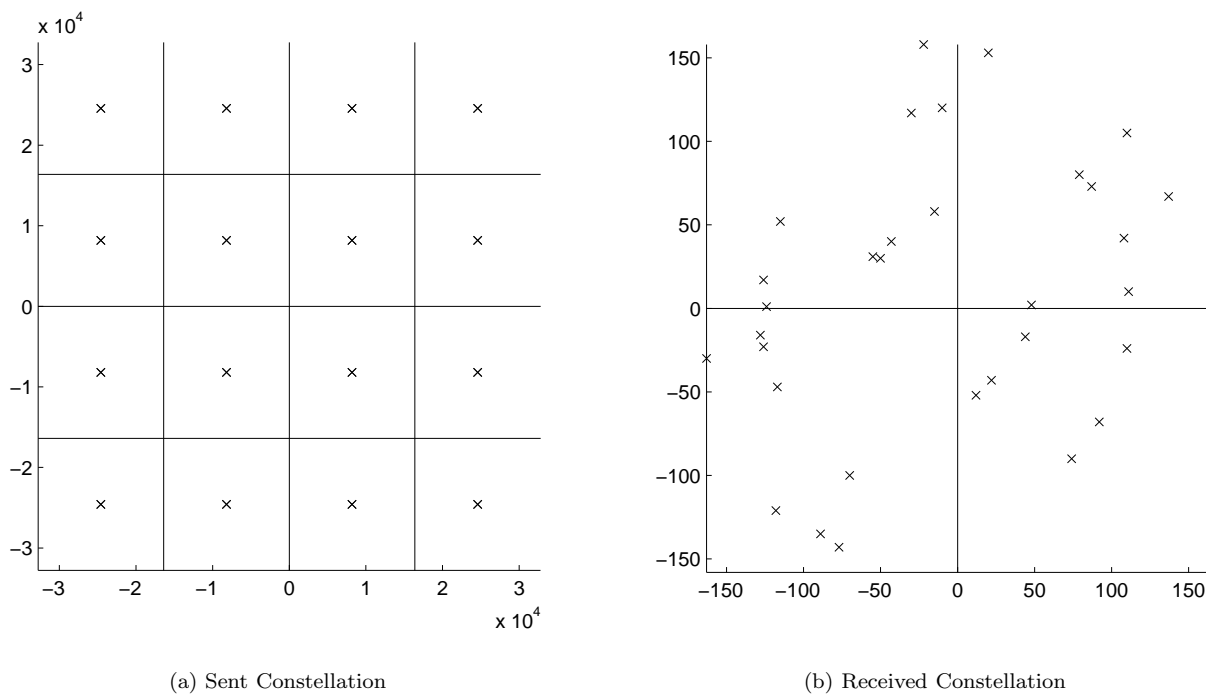


Figure 7: The channel estimation frame consists of each 16-QAM constellation points enumerated into the carriers, one point per carrier, in an arbitrary but known order.

shift and scale the symbols as required to negate the combined effects of the channel.

Figure 8 shows the inverse of the estimated channel response. From the phase plot it can be seen that the channel noise effects reduce to a matter of phase shift. This phase shift, which manifests as a rotation on the 16-QAM constellation, is accounted for by the Channel Adjustment Matrix.

3.3 Header Frame

The header frame contains information about the number of data frames that follow. This information is required to ensure that the correct amount of data is received. The frame also contains redundant information so that symbol errors do not affect the operation of the modem. Since we are not employing any error correction codes, it is very likely that a few of the values received will be incorrect. Since this is very important information, it is placed in the frame several times. The periodicity caused by this multiple repetition of information posed a clipping problem in the time domain. To counter this problem, the header information was exclusive-or'ed with a random sequence and the resulting non-periodic information was modulated and transmitted. At the receiver the original header content is retrieved by exclusive-or'ing again with the same known sequence.

The received constellation plot of the header frame is depicted in Figure 9(a).

The header frame also goes through deterioration in the channel resulting in a disoriented and skewed constellation. The Channel Adjustment Matrix formed from the channel adjustment frame is then employed to correct the constellation. The adjusted header frame constellation is shown in Figure 9(b).

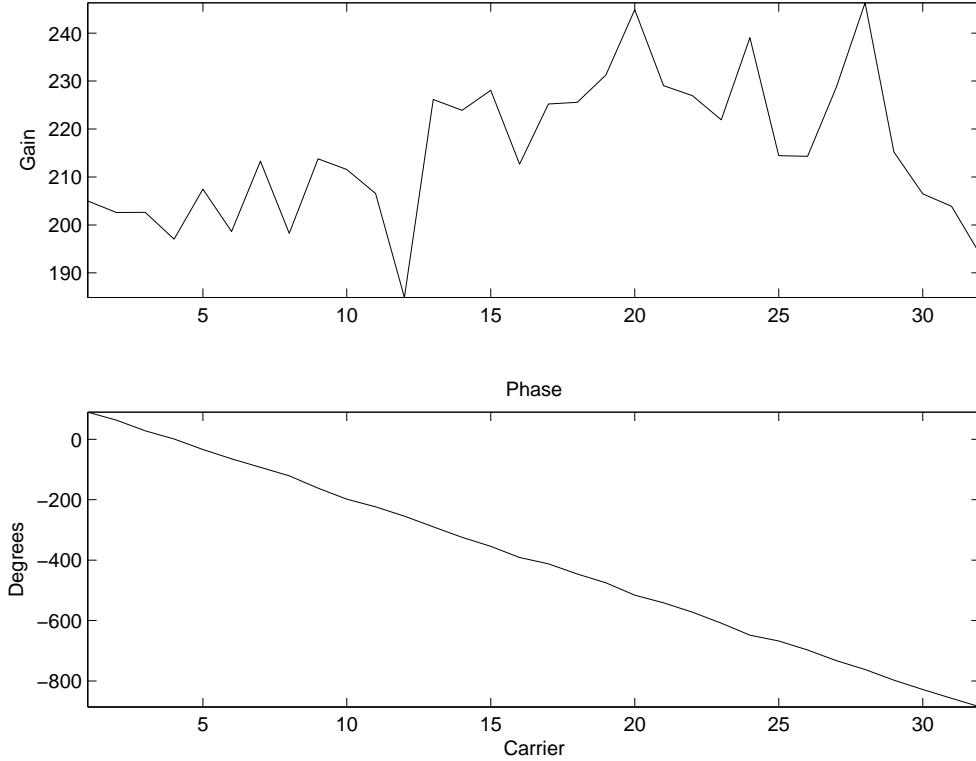


Figure 8: Compensator transfer function for estimated channel response.

4 Implementation

The modem is full duplex; that is, it can send and receive data simultaneously. It is important that the modem concurrently performs both tasks. By the use of buffering, they are allowed to run in an interleaved series. However, each task must have a chance to run periodically, so that the buffers do not overflow with unconsumed data. The serial and analog interfaces transceive data through interrupts triggered for this purpose. Data transceived on the serial interface is placed into packets. The amount of data in each packet is the amount of data available for transmission when the header frame is sent, capped at a preset limit.

4.1 Packet Structure

The packet structure illustrated in Figure 10 above was implemented in our project. The total packet length is $3427 + 1224k_1$ samples, as shown in Table 2, where k_1 denotes the number of data frames sent.

4.2 Implemented Modules

The project was divided into four main modules:

- UART
- OFDM

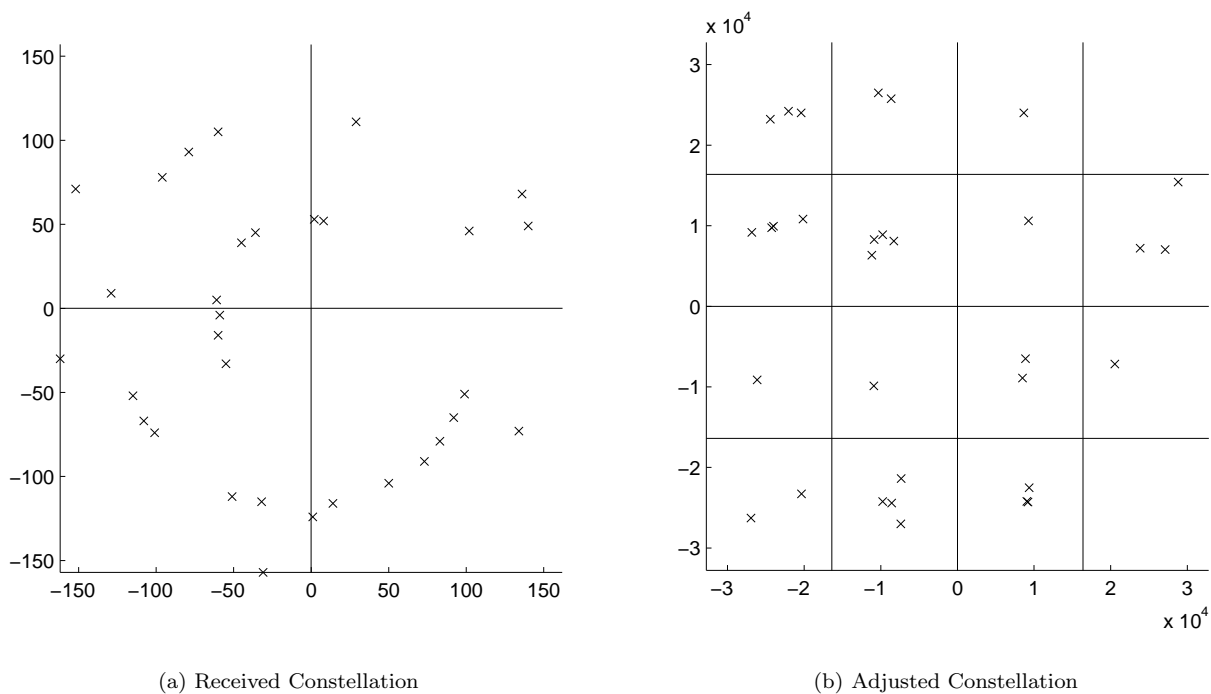


Figure 9: The header frame consists of bytes representing the amount of data to be sent in the current packet, exclusive-or'ed with a randomizing one-time-pad, modulated by 16-QAM.

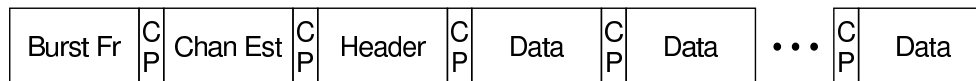


Figure 10: Data packet structure, showing the cyclic prefix (CP) prepended to the data-containing frames.

- IF / CODEC
- CORR

These modules are used for both transmitter and receiver operations, and are explained in the following sub-sections.

4.2.1 UART Module

The Universal Asynchronous Receiver-Transmitter (UART) is used to handle asynchronous serial communication. In our project it is used to send data between the computer and the C5510 DSK where it is processed and transmitted.

The UART module uses the Recommended Standard 232 (RS-232) interface and communicates between different nodes and devices. It performs the parallel to serial conversion of the digital data that is transmitted and the serial to parallel conversion of digital data that has been received. The baud-rate, number of stop

| <i>Component</i> | <i>Length (without CP)</i> | <i>Cyclic Prefix</i> | <i>Total Length</i> |
|--------------------------|----------------------------|----------------------|---------------------|
| Burst Frame | 1024 | No | 1024 |
| Channel Estimation Frame | 1024 | Yes | 1224 |
| Header Frame | 1024 | Yes | 1224 |
| Data Frame | 1024 | Yes | 1224 |
| Packet | | | $3472 + 1224k_1$ |

Table 2: Makeup, in samples, of components of each packet. k_1 denotes the number of data frames sent.

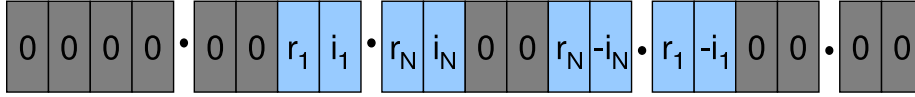


Figure 11: Placement of symbols into IFFT bins.

bits, and number of data bits can be set as necessary. In our project, a baud rate of 115,200 was used to minimize latency.

We implemented hardware flow control to keep the buffer-fill level to a size less than allocated. Transmit and receive buffers are handled by this module, and it ensures that data that needs to be sent is transmitted as soon as possible.

4.2.2 OFDM Module

The OFDM module is the main module which uses the other modules. It takes data from the UART receive buffer in byte form. Then the bits are taken four at a time and are modulated using 16-QAM. A Gray coded constellation of 16-QAM, shown in Figure 3 and Table 1, is used to minimize the errors caused by high variance noise. Conceptually, these symbols represent the frequency domain coefficients.

The symbols generated are then processed by an Inverse Fast Fourier Transform operation. Since our data is real-valued, and we want a real result at the output, we take the flipped replica of the data and append its conjugate to the original data, exploiting the Fourier transform properties. As a result real data is seen at the output which is in time domain. Since most communication systems are considered to be AC coupled, data at DC (very low frequencies) must be avoided. To cope with this, zeroes are inserted at appropriate places in the IFFT bins to eliminate any DC component at the output and in between data sets. Also, zeros are placed as necessary into the bins between the data to form a power-of-two number of bins (necessary for efficient IFFT computation). The structure of the set of bins is shown in Figure 11. The list contains two entries for each coefficient because the real and imaginary values are interleaved.

The OFDM module is also responsible for generating the packet that was shown in Figure 10. A variable number of data frames can be present in a single packet. At the transmitter end, the time domain packet is sent to the IF / CODEC module where it is prepared to be sent across the channel.

Similar actions take place on the receiver side, where the FFT is the main operation followed by 16-QAM demodulation. The same module is responsible for detection of burst frame, evaluation of channel estimation frame, generation of the channel adjustment matrix, adjustment of the following frames, processing of the header frame and keeping a count of the expected number of frames to follow.

4.2.3 IF / CODEC Module

This library handles the functions of AIC-23 CODEC, which is responsible for digital to analog conversion on the transmitting side, and for analog to digital conversion on the receiving side. Circular buffers were used, one each for the transmitter and receiver. These buffers were then temporarily point-linearized to facilitate processing. The interface to this module is described in Section 4.6.2.

4.2.4 CORR Module

The CORR (correlation) module is responsible for the critical process of time synchronization. The idea is to implement a running correlation algorithm such that it detects the burst frame prepended to an incoming packet with high probability. The module calculates the normalized correlation of the burst frame with incoming data. The correlation value is compared to a pre-computed threshold, which in our case is 40% of the maximum normalized correlation.

Since the burst frame is a pseudorandom signal, it has one distinctive peak of correlation at zero delay. The clock safety adjustment places the beginning of the FFT data set approximately in the middle of the cyclic prefix. Therefore, if the burst frame is detected at any of the near-center autocorrelation peaks, the FFT will still be taken on data in the correct frame. Without this adjustment, part of the FFT data set could belong to the following frame's cyclic prefix, which would cause poor results.

4.3 Transmitter Implementation

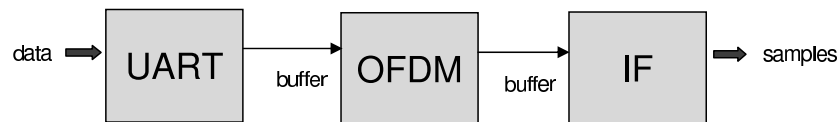


Figure 12: Data flow path for sending data through the modem.

As shown in Figure 12, the UART module converts parallel data in the computer into serial data and transmits it to the DSK that receives the data from the computer, and feeds it to the OFDM module.

In the OFDM Module, data bits received from the UART module are modulated using 16-QAM with 4 bits combined into one symbol. Packets are then passed to the D/A output, where they are transmitted over the channel.

The transmitter working can be explained with the state diagram shown in Figure 13. It starts in the Wait state, where the transmitter remains until it receives input from the UART. The received input is then processed by the OFDM module. The burst frame is sent, followed by the channel estimation frame. Then the header is sent which is followed by the data frame. The transmitter keeps sending the bytes until all the data has been transmitted, packing with random bits where necessary. At the end the transmitter returns to the Wait state, waiting for more data.

4.4 Receiver Implementation

Signal samples are taken into DSK from the A/D buffer. The burst frame is then used to detect the arrival of a packet and is critical for time synchronization. The Channel Estimation frame is used to estimate the channel response, the channel adjustment matrix is generated and the incoming data is scaled appropriately.

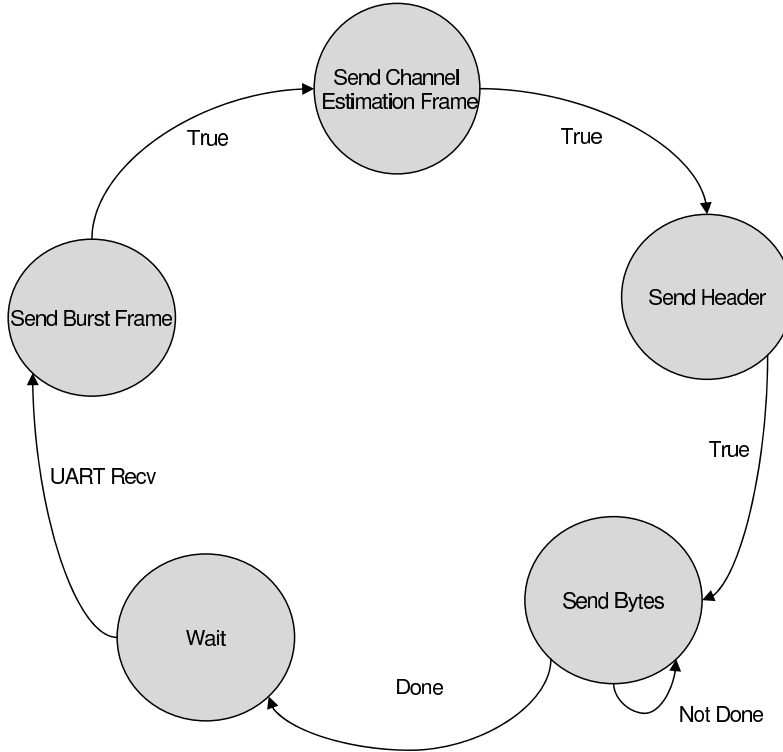


Figure 13: State machine for sending data through the modem.

The FFT is taken on the samples, which represents the cyclic prefix and frame, in accordance with the clock safety skew. The 16-QAM demodulation is the last step, which is implemented using minimum-distance criterion on received samples. Data bytes are then passed to the computer using serial UART link. Figure 14 illustrates the Receiver Data Flow.

The state transition diagram of the receiver is shown in Figure 15. The receiver starts in the Wait and Check states, where it constantly computes the correlation of the received data with the burst frame, as soon as there are enough samples received. If the burst frame is not detected, the receiver consumes some samples and returns to the Wait state. When the burst frame is detected, the receiver moves to the next state to receive the channel estimation frame followed by the header. If the header reception is successful (the header content is consistent) then the receiver moves to the data reception mode and remains there until all the data is received. When the number of bytes mentioned in the header frame are received, the state changes to Wait again, where it starts checking for the arrival of a new burst frame.

4.5 Hardware Design Issues

We had to decide upon following hardware and protocol issues, which were chosen to suit our approach, the hardware available, and the empirical feedback.

DSP Global’s DSPG-IC2-U232 serial daughter-boards were used to communicate with the serial port of the computers on both the transmitting and receiving ends. We used 4 bits to form a symbol which was then QAM modulated. The number of sub-channels we used was 32 which resulted in 16 bytes per data frame. A cyclic prefix of 200 samples was used in addition to a clock safety of 70. The packet size is variable and is

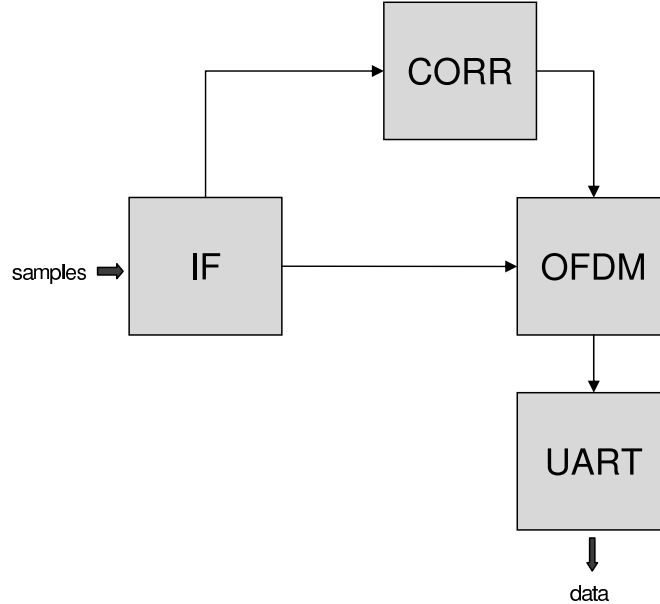


Figure 14: Data flow path for receiving data from the modem.

set on the fly depending on the amount of data to be sent. If the data is not sufficient, then the transmitter waits a few milliseconds for any additional data; however in the case where no more data is received, the timer times out and the data is then sent.

4.6 DSP Implementation issues

4.6.1 Math Processing

We modeled our OFDM system in MATLAB, which is a 64-bit package; however our hardware implementation was on a fixed point processor where all the buffers were chosen to be 16-bits wide. The correlation values exceeded the upper limit of 32 bit registers and we had to resort to using the 40 bit registers. Most of this processing was explicitly handled in assembly language.

4.6.2 IF Module

The IF interface code was derived from an assembly source file provided to us. The provided file was designed to support interrupt-driven, buffered digital-to-analog conversions. This library was adequate for our purposes, but we added functionality to optimize the code which uses this library. Table 3 shows the interface differences. For instance, the function to copy data into the output buffer, and the function to copy data from the input buffer, were changed to take vectors and a vector stride as arguments. The original functionality, which moved one sample at a time, resulted in overhead-intensive loops calling these functions repeatedly. By taking a vector of samples, the number of instructions inside the loop was greatly reduced. A common operation was to use A/D samples as input to an FFT function, which takes interleaved (Real, Imaginary) data. By setting the vector stride parameter to two, the IF functions will effectively place data into the real components of each bin. Again, this resulted in a very large saving of operations inside the loop.

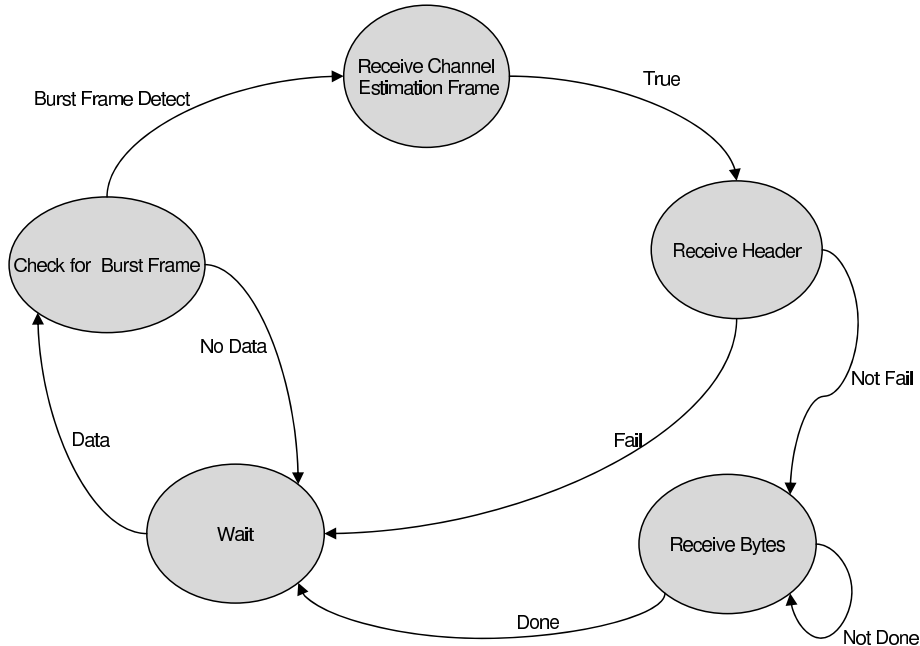


Figure 15: State machine for receiving data from the modem.

| IF Module | CODEC Module |
|---|--|
| void ifSendSetup(short *buffer, short length); | Mc2X_put_setup(short *buffer, short length); |
| void ifRecvSetup(short *buffer, short length); | Mc2R_get_setup(short *buffer, short length); |
| void ifSend(short *buffer, short length, short stride); | Mc2X_put(short left, short right); |
| short ifSendCount(); | |
| void ifRecv(short *buffer, short length, short stride); | Mc2R_get(short *left, short *right); |
| short ifRecvCount(); | |
| short *ifRecvLinearize(short length); | |

Table 3: Functions supported by the revised IF module, and function supported by the original CODEC module.

The main functions of these interfaces are to setup the codec for proper functionality, including the sampling rate, which in our case was 32,000 samples per second. It also keeps the count of received and transmitted samples and manages the related buffers.

4.6.3 UART Module

Also provided to us was a C support library for the UART interface board. The interrupt service routine provided with this library was not written correctly, nor did it support RS-232 flow control. The interrupt code must service all UART interrupts routed through the INT1 hardware interrupt before it returns. Otherwise, the hardware interrupt does not reset, and the service routine does not get called again because it is edge triggered.

We modified the provided code, and named it the UART library. In the interrupt, when the UART receive buffer is nearly full, the library deasserts the Clear-to-Send (CTS) line. This causes the computer to stop

sending data until this line is reasserted. In the routine which copies the serial data to the application's buffers, when sufficient data has been removed from the buffer, the CTS line is automatically reasserted. By using two thresholds, the buffer is hysterically filled. Because the modem cannot transceive data as quickly as the computer and modem can communicate with each other, flow control is necessary. Otherwise, the modem would lose data when its transmit buffers became full. As with the IF library, functions in the modified UART library take vectors and vector strides as arguments. With these modifications, the UART library met the modem's needs.

4.6.4 Optimizations

As mentioned above, a running correlation is used to detect the burst frame. Because the data is stored in a large circular buffer, using in-place correlation of the received data is difficult. To this end, the `ifRecvLinearize` function ensures that the desired number of samples can be found in linear, contiguous address spaces in memory, and returns the base address of this set of samples. For requested data sets which do not wrap from the end to the beginning, it simply checks for the existence of at least the correct number of samples, and then returns the starting address. Then, correlation can proceed on this pointer. For data sets which will wrap, the correct number of samples are copied from the beginning of the buffer into a padding space past the end of the circular buffer. In this manner, the returned pointer is to the base of a linear, contiguous data set. The maximum number of words copied is one less than the number desired, and a copy of this, and all sizes, is infrequent. The linearization process of the receive circular buffer is shown in Figure 16.

To perform the correlation, we modified a convolution function provided with TI's DSP library. At the heart of this code is a assembly RPT/MACM loop, which executes N instructions to correlate two length- N vectors. This is as fast as is possible, in the general case, and is much faster than when written in C. Since the correlate function is called so frequently, a short execution time is necessary. When combined with the circular buffer linearization, burst frame detection can be done extremely quickly.

5 Results and Findings

Figure 17 is the plot of the Bit Error Rate vs. the SNR for our system obtained in MATLAB under simulated conditions. It can be noticed that the BER is very high at low values of SNR and gradually decreases as the SNR increases. It can also be noticed that there is a steep decline in BER at the 6dB mark. This indicates that at lower SNR values, synchronization is very sensitive to noise causing errors in frame detection at the receiving end.

We have successfully implemented a compile-time configurable OFDM modem on a Texas Instruments TMS320C5510 DSP, fully capable of full-duplex operation, at speeds between 5-8 Kbps with over 90% data accuracy.

5.1 Improvements

We would like to program the modem binary into the DSK Flash memory, which would let it operate in a stand-alone manner. Most of the useful applications of DSPs are stand-alone and contain the related software on Flash memory, from where it is loaded onto the program memory for execution.

Other improvements include manual conversion of some of the C code into assembly, for faster execution. We believe that a reasonable speed-up could be achieved following this enhancement.

On the algorithmic side, we would like to use an adaptive-step correlation algorithm. Being the most important step in packet detection and time synchronization, we cannot avoid correlation even though it

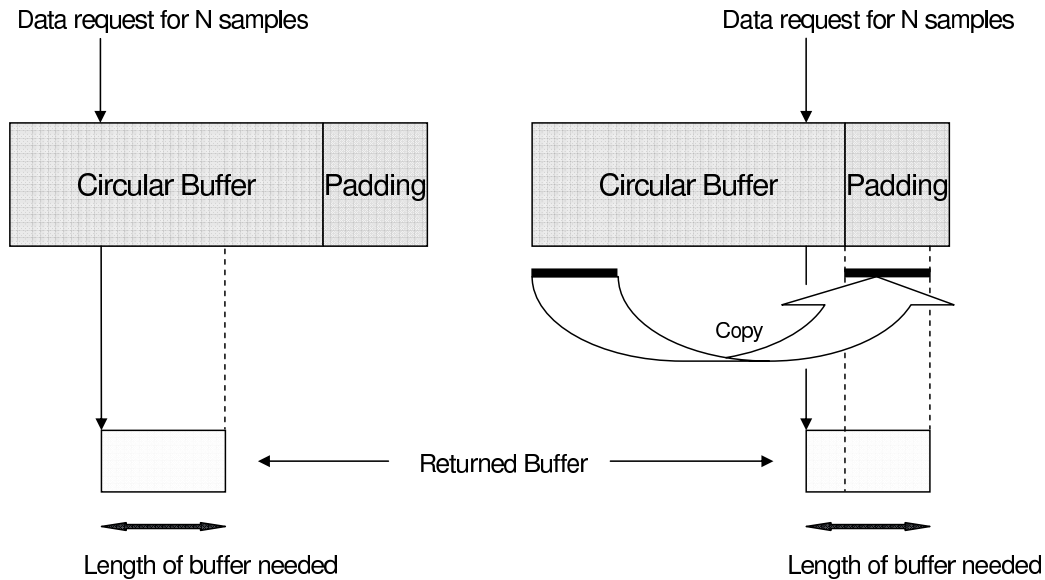


Figure 16: The circular buffer is a large buffer and we are interested only in N samples of the received signal. In the first case, the request is towards the start of the circular buffer, and since N samples of the data are placed contiguously and in the order in which they were received, any extra processing need not be performed. If the request begins from a point such that the data would not remain contiguous for N samples, like the second case, the number of data samples required to form N contiguous samples are copied from the front of the buffer into the padding zone.

is highly computationally intensive to calculate a new correlation for every incoming sample taken into the frame. However, using adaptive-step correlation we can skip over more samples when the likelihood of a burst frame being nearby is remote. Changing to adaptive correlation would have multifold benefits: the computation necessary would reduce, allowing more time for other tasks. This which would translate to smaller power drain, a desirable trait for wireless devices.

Inclusion of standard error correction codes is also expected to improve the accuracy by orders of magnitude. However, channel bandwidth is used for adding redundant data, and this will reduce the capacity of the system.

6 Commercial Markets and Uses

Orthogonal Frequency Division Multiplexing is bandwidth efficient in that it uses orthogonal frequencies to counter Inter-Carrier Interference (ICI) and, as shown in the discussion above, conserves bandwidth by placing the channels closer than conventional frequency division multiplexing schemes.

It is also robust in Rayleigh fading, multipath propagation and Inter-Symbol Interference (ISI), because of the fact that it uses multiple carriers to transmit information. Since each channel has smaller data rate, it results in larger symbol durations, leading to smaller or negligible ISI.

The main aim of the project was to develop the basic structure of an OFDM capable modem, which could be used for further experimentation in the field of modern communications. Our project is, by all standards, a good learning tool for anybody who seeks to gain insight into this promising technology of present and future communication systems. We have created a test bench for both the technically sound and non-technical

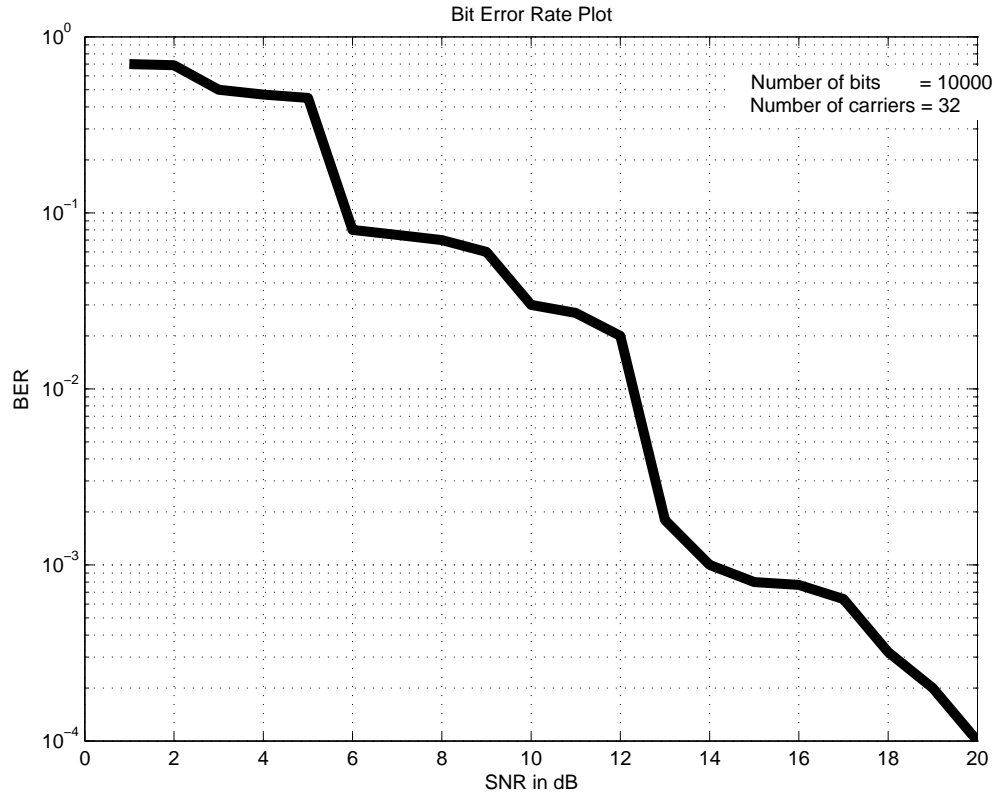


Figure 17: Plot of Bit Error Rate (BER) vs SNR for our model.

pupil, who would like to learn the basic functionality of such a modem, or would like to understand the requirements, limitations, pros and cons of choosing this approach over any other, when used independently or in conjunction with some other device.

Our implementation of Orthogonal Frequency Division Multiplexing is flexible enough to give the designer the independence of choosing the number of channels or sub carriers. It also lets the user experiment with custom cyclic prefix widths, so that differences in performance can be measured. Together with our MATLAB simulations, the entire package gives the students a feel for the performance differences between ideal scenarios of 64 bit word sizes on a gigahertz processor, versus the 16 bit word size of a 200 MHz processor.

While the modem gives a fair enough performance without the use of any error correction, we can enhance its performance with the use of simple or more complex error correction codes before launching into the market.

Another area of importance is the channel estimation. Channel estimation has haunted scientists for a long time, and there does not exist any best approach to implement this task. Estimation of the response for a time variant or invariant channel, and taking corrective measures at the receiver to ensure proper reception, is one of the most important components of a communication protocol. As our discussion and graphs proved, without channel estimation a reasonable communication system is not possible. The users have the freedom to change the estimation algorithm according to their choice. We have used a scheme which is based on the training / learning sequences. It can be altered and better schemes could be incorporated to measure any differences in performance.

OFDM is set to be the choice of designers for future communication. The IEEE 802.11a and 802.11g use OFDM, which is used for wireless LANs. The Asynchronous Digital Subscriber Line (ADSL) is OFDM based, and so is the Digital Audio-Video Broadcast (DAB/DVB).

7 Conclusion

There were numerous challenges in completing this project. Staying current with the timeline chosen was the most difficult task. The project scheduling became backweighted, and this resulted in the majority of the work being done at the end.

However, the modem implementation was successful. We have transmitted and received data, and have sent varying amounts of data in a packet. This fully stresses the capabilities of the buffering and math processing. Our modem is fully capable of full-duplex operation at speeds of 5-8 Kbps.

As a proof of concept, this project was highly successful, in both showing the flexibility and robustness of Orthogonal Frequency Division Multiplexing. It is recommended that this project be further pursued. The incorporation of error correcting codes will hugely improve the performance of the system in terms of accuracy. The use of an adaptive-step correlation algorithm will decrease the processing power required, at almost no expense in performance.

References

- [1] Marc Engels, Wireless OFDM Systems-How to make them work?
- [2] Dusan Matiae, Introduction to OFDM, II Edition.

Glossary

1. *Bandwidth:*
The amount of data that can be sent over a connection in a given period of time. Bandwidth is usually stated in bits per second (bps). Also, the amount of frequency band used by a communication system; usually measured in Hertz (Hz).
2. *Baseband:*
Name given to a transmission method in which the entire bandwidth is used to transmit just one signal.
3. *BER:*
Acronym for Bit Error Rate. In a digital transmission, BER is the percentage of bits with errors divided by the total number of bits that have been transmitted or received or processed over a given time period. The rate is typically expressed as 10 to the negative power. For example, four erroneous bits out of 100,000 bits transmitted would be expressed as 4×10^{-5} .
4. *Carrier:*
A high frequency waveform that is modulated (modified) to represent the information or data to be transmitted.
5. *Demodulation:*
The process of recovering a modulating signal from a modulated carrier.

6. *Digital Communication Systems:*
A system that transmits and receives information that can be represented as a stream of bits (BInary digiTs 1 and 0s).
7. *FFT:*
Acronym for Fast Fourier Transform. An algorithm for fast computation of the Fourier transform of a set of discrete data values.
8. *Flash Memory:*
A non-volatile memory device that retains its data after the power is removed.
9. *Frequency-Division Multiplexing:*
A scheme in which numerous signals are combined for transmission on a single communications line or channel. Each signal is assigned a different frequency (subchannel) within the main channel.
10. *Full-Duplex:*
The ability of a communication system to transport data in both directions simultaneously.
11. *Gray Coding:*
A Gray code is a binary number system where two successive values differ in only one digit. The code was designed by Bell Labs researcher Frank Gray and patented in 1953.
12. *ICI:*
Acronym for Inter-Carrier Interference. Undesirable phenomenon of energy interference between different symbols in a channel.
13. *IFFT:*
Acronym for Inverse Fast Fourier Transform. An algorithm for fast computation of the Inverse Fourier transform of a set of discrete data values.
14. *ISI:*
Acronym for Inter-Symbol Interference. Undesirable phenomenon of energy interference between different symbols in a channel.
15. *Modem:*
Short for modulator/demodulator. A communication device that converts one form of a signal to another such that it is suitable for transmission over a communication channel; typically from digital to analog and then from analog to digital.
16. *Modulation:*
Coding of information onto the carrier frequency. This includes amplitude, frequency, or phase.
17. *Multipath:*
The problem caused by multiple copies of the same signal arriving at the receiver simultaneously via different propagation paths. Signals that are in phase will add to one another. Signals that are out of phase will cancel one another.
18. *OFDM:*
Acronym for Orthogonal Frequency-Division Multiplexing. A transmission technique based on Frequency-Division Multiplexing (FDM) where multiple signals are sent out at different orthogonal frequencies.
19. *Orthogonality:*
The property shared by two factors that ensures that one factor can be evaluated without considering the other factor to which it is orthogonal.
20. *Packet:*
A group of bits transmitted as a unit.

21. *QAM*:
Acronym for Quadrature Amplitude Modulation. A modulation technique which uses amplitude as well as phase for encoding data to achieve higher data rates.
22. *Rayleigh Fading*:
Multipath effects characterized by the Rayleigh Distribution.
23. *RS-232*:
Acronym for Recommended Standard 232. This is the standard for communication through PC serial ports.
24. *SNR*:
Acronym for Signal-to-Noise ratio. The relationship between the useful signal and extraneously present noise, usually expressed in dB.
25. *UART*:
Acronym for Universal Asynchronous Receiver Transmitter. The UART is a computer component that handles asynchronous serial communication. Every computer contains a UART to manage the serial ports, and all internal modems have their own UART.
26. *XOR*:
Acronym for eXclusive-OR. A logical operator that results in true if one of the operands, but not both of them, is true.

Appendix A

ofdm.c

```
1 #include <dsplib.h>
2 #include <stdio.h>
3 #include "support\McBSP_452.h"

5 void cfft32_SCALE(LDATA *x, ushort nx);
6 void cbrev32 (LDATA *x, LDATA *y, ushort n);
7 int ltoa(long val, char *buffer);

9 /* define timer0 registers */

11 #define TIM0 (*(ioport unsigned *)0x1000)
12 #define PRD0 (*(ioport unsigned *)0x1001)
13 #define TCR0 (*(ioport unsigned *)0x1002)
14 #define PRSC0 (*(ioport unsigned *)0x1003)
15
16 /* define bit fields for TCR */
17
18 #define T_IDLEEN 0x8000
19 #define T_INTTEX 0x4000
20 #define T_ERRTM 0x2000
21 #define T_FUNC 0x0800
22 #define T_TLB 0x0400
23 #define T_SOFT 0x0200
24 #define T_FREE 0x0100
25 #define T_PWD 0x0040
26 #define T_ARB 0x0020
27 #define T_TSS 0x0010
28 #define T_CP 0x0004
29 #define T_POLAR 0x0002
30 #define T_DATOUT 0x0001
31

33 ioport unsigned *CLKMD=(ioport unsigned *)0x1c00;

35 void CPUinit(int pll_mult, int pll_div);

37 /* set pll multiplier, divider, and enable pll.
38    does not return until pll locked. */
39 void CPUinit(int pll_mult, int pll_div)
40 {
41     unsigned new_clkmd;
42
43     new_clkmd = *CLKMD;

44     /* new clock frequency is mult/(div+1)*input_clock */
45     new_clkmd &= ~((0x1F << 7) | (0x2 << 5));
46     new_clkmd |= ((pll_mult & 0x1F) << 7) | ((pll_div & 0x2) << 5) | (0x1 << 4);

47     /* set new register values all at once, and wait for lock */
48     *CLKMD = new_clkmd;
49     while ((*CLKMD & 0x1) == 0)
50         ;
51 }

53

55 #define FTV_CNT 33
56 #define FS 48000

59 /* IF.asm */
60 short ifRecvCount();
61 void ifRecv(short *, short, short);
62 void ifRecvSetup(short *, short);
63 short *ifRecvLinearize(short);
64 short ifSendCount();
65 void ifSend(short *, short, short);
66 void ifSendSetup(short *, short);

69 /* UART.c */
70 void uartSetup(short *, int, short *, int, int);
71 short uartRecvCount();
```

```

void uartRecv(short *, short, short);
73 short uartSendCount();
void uartSend(short *, short, short);
75
/* ofdm.c */
77 void packBytes(DATA *bytes, DATA *symbols);
void packSymbols(DATA *s, DATA *symbols);
79 void sendSymbols(DATA *symbols);
void unpackBytes(DATA *bytes, DATA *symbols);
81 void recvSymbols(DATA *symbols);
unsigned long ComputeFTV(unsigned long, unsigned long);
83
/* corr3.asm */
85 DATA corr3(DATA *x, DATA *y, int length);

87 /* respectively, aic input buffer size, output, serial input, output */
#define AIBUF_SIZE      0x3000//8192
89 #define AOBUF_SIZE    0x3000//4096
#define UIBUF_SIZE      2048
91 #define UOBUF_SIZE    2048

93 #define N              32                /* # symbols in frame, must have integer number of
      BPS-symbols */
#define BLS             20                /* baseline-shift — number of bins to discard at
      the bottom of the fft */
95 #define FFT_LEN       1024             /* # samples in frame, must be >= 2*N+2 */
#define GDI             200              /* guard delay samples, must be <= N */
97 #define BPS           4                /* bits per symbol, must be 4 */
#define BPF             (N*BPS/8)        /* bytes per frame, must be multiple of 4 */
99 #define CKS           70              /* clock safety factor, must be >= 0 */
#define MAX_FR          20              /* max number of frames in packet, must be >= 1 */
101 #define HD_AGG        1               /* header aggressiveness */

103 #pragma      DATA_SECTION(aicInput, "aicinput")
#pragma      DATA_ALIGN(aicInput, 2)
105 #pragma      DATA_SECTION(aicOutput, "buffers2")
#pragma      DATA_ALIGN(aicOutput, 2)
107 #pragma      DATA_SECTION(uartInput, "buffers")
#pragma      DATA_ALIGN(uartInput, 2)
109 #pragma      DATA_SECTION(uartInput, "buffers")
#pragma      DATA_ALIGN(uartOutput, 2)
111
DATA aicInput[AIBUF_SIZE+FFT_LEN+2], aicOutput[AOBUF_SIZE];
113 DATA uartInput[UIBUF_SIZE], uartOutput[UOBUF_SIZE];

115 /* QAM16 encoding */
/*
117 construct a grey-coded qam constellation with the
number of bits set in each word following:
119     0 1 2 1
121     1 2 3 2
123     2 3 4 3
125     1 2 3 2
127 and we get:
0000 0001 1001 1000
125 0010 0011 1011 1010
0110 0111 1111 1110
127 0100 0101 1101 1100
*/
129 DATA QAM16GCr[16] = {-3, -1, -3, -1, -3, -1, -3, -1, 3, 1, 3, 1, 3, 1, 3, 1},
QAM16GCi[16] = { 3, 3, 1, 1, -3, -3, -1, -1, 3, 3, 1, 1, -3, -3, -1, -1},
131 QAM16GCd[16] = {8, 10, 14, 12, 9, 11, 15, 13, 1, 3, 7, 5, 0, 2, 6, 4};

133 #define XDT1      0x4000 /* must be in decreasing order */
#define XDT2      0
135 #define XDT3     -0x4000
#define YDT1      0x4000 /* must be in decreasing order */
137 #define YDT2     0
#define YDT3     -0x4000
139
/* globals — use may vary depending on tx/rx */
141 DATA chanAdjMat[N][4];
DATA syms[2*(GDI+FFT_LEN)];
143 DATA bfs[2*FFT_LEN];
LDATA newBuf[2*(FFT_LEN+GDI)];
145
#define CONS      40

```

```

147 DATA const1[CONS][CONS];

149 DATA otp[BPF];
    DATA temp[2*N];
151 DATA BURST_FR[2*N];
    DATA CHAN_DET[2*N];
153 DATA failcnt;
    short bfd;
155 DATA gr, gs, corout, *dptr;
    LDATA cor, corval;
157 LDATA corsqr, corsqr2;

159 extern unsigned long ftvR, ftvS, DDSaccumR, DDSaccumS;

161 /* 256 value sine table */
    signed SineTable[256] = {
163         0,      804,    1608,    2410,    3212,    4011,    4808,    5602,
165         6393,    7179,    7962,    8739,    9512,    10278,   11039,   11793,
167         12539,   13279,   14010,   14732,   15446,   16151,   16846,   17530,
169         18204,   18868,   19519,   20159,   20787,   21403,   22005,   22594,
171         23170,   23731,   24279,   24811,   25329,   25832,   26319,   26790,
173         27245,   27683,   28105,   28510,   28898,   29268,   29621,   29956,
175         30273,   30571,   30852,   31113,   31356,   31580,   31785,   31971,
177         32137,   32285,   32412,   32521,   32609,   32678,   32728,   32757,
179         32767,   32757,   32728,   32678,   32609,   32521,   32412,   32285,
181         32137,   31971,   31785,   31580,   31356,   31113,   30852,   30571,
183         30273,   29956,   29621,   29268,   28898,   28510,   28105,   27683,
185         27245,   26790,   26319,   25832,   25329,   24811,   24279,   23731,
187         23170,   22594,   22005,   21403,   20787,   20159,   19519,   18868,
189         18204,   17530,   16846,   16151,   15446,   14732,   14010,   13279,
191         12539,   11793,   11039,   10278,   9512,    8739,    7962,    7179,
193         6393,    5602,    4808,    4011,    3212,    2410,    1608,    804,
195         0,      -804,   -1608,   -2410,   -3212,   -4011,   -4808,   -5602,
197         -6393,   -7179,   -7962,   -8739,   -9512,  -10278, -11039, -11793,
199         -12539, -13279, -14010, -14732, -15446, -16151, -16846, -17530,
201         -18204, -18868, -19519, -20159, -20787, -21403, -22005, -22594,
203         -23170, -23731, -24279, -24811, -25329, -25832, -26319, -26790,
205         -27245, -27683, -28105, -28510, -28898, -29268, -29621, -29956,
207         -30273, -30571, -30852, -31113, -31356, -31580, -31785, -31971,
209         -32137, -32285, -32412, -32521, -32609, -32678, -32728, -32757,
211         -32767, -32757, -32728, -32678, -32609, -32521, -32412, -32285,
213         -32137, -31971, -31785, -31580, -31356, -31113, -30852, -30571,
215         -30273, -29956, -29621, -29268, -28898, -28510, -28105, -27683,
217         -27245, -26790, -26319, -25832, -25329, -24811, -24279, -23731,
219         -23170, -22594, -22005, -21403, -20787, -20159, -19519, -18868,
221         -18204, -17530, -16846, -16151, -15446, -14732, -14010, -13279,
223         -12539, -11793, -11039, -10278, -9512,   -8739,   -7962,   -7179,
225         -6393,  -5602,  -4808,  -4011,  -3212,  -2410,  -1608,  -804,
    };

197 /* for data-send timeout */
    void resetTimer(void)
199 {
201     TCR0 = T_TSS|T_TLB|T_FREE|T_ARB; // stop, load, enable reload
203     PRSC0 = (8-1)*0x0041; // prescaler set to 1
205     PRD0 = 0xFFFF; // initialize to max count
207     TCR0 = TCR0&^(T_TSS|T_TLB); // clear loading and start counting
209 }

207 /* OFDM main loop, performs tx and rx */
209 void main(void)
    {
211     unsigned short i, j, ii, mask;
213     short junk, rxMode, txMode;
215     DATA re, im, res, ims, g;
217     unsigned long rsd, rmd, rsc;
219     unsigned short send_prog, send_count;
221     unsigned short recv_prog, recv_count;
223     short majCntL, majCntH;
225     short buf[10];
227     unsigned long cc;
229     unsigned input;
231     unsigned long dip_led = 0x300000;

233 /* initialize CPU clock speed */

```

```

CPUinit(25, 2);
225 /* initialize send timer */
227 resetTimer();

229 /* initialize codec using setup_codec.c */
ifRecvSetup(aicInput, AIBUF_SIZE);
231 ifSendSetup(aicOutput, AOBUF_SIZE);
setup_codec();
233 McBSP_send(1, 8*0x0200+0x0019);
startup();
235

/* initialize serial uart using UART2support.c */
237 uartSetup(uartInput, UIBUF_SIZE, uartOutput, UOBUF_SIZE, 2/*12*/);

239 failcnt = 0;
bfd = 0;
241 corval = 400;

243 /* build QAM-16 grey-coded table */
for (i = 0; i < 16; i++)
245     {
        QAM16GCr[i] = QAM16GCr[i] << 13;
247     QAM16GCi[i] = QAM16GCi[i] << 13;
    }
249
srand(11);
251 /* create burst and channel estimation frames */
for (i = 0; i < N; i++)
253     {
        if (i < 16)
255         {
            BURST_FR[2*i] = rand()/2;//1<<(i%16);
257             BURST_FR[2*i+1] = rand()/2;
        }
259         else
            {
261             BURST_FR[2*i] = 0;
                BURST_FR[2*i+1] = 0;
263             }
        CHAN_DET[2*i] = QAM16GCr[i%16];
265         CHAN_DET[2*i+1] = QAM16GCi[i%16];
    }
267
for (i = 0; i < BPF; i++)
269     otp[i] = rand()&0xFF;

271 /* must implement band-pass filtering before changing this upmixing value */
ftvR = ftvS = ComputeFTV(0, 48000);
273 DDSaccumR = DDSaccumS = 64<<24;

275
/* set up burst frame for correlation against */
277 packSymbols(BURST_FR, syms);
cifft(syms, FFT_LEN, SCALE);
279 cbrev(syms, syms, FFT_LEN);
for (i = 0; i < FFT_LEN; i++)
281     bfs[i] = 40*syms[i*2];
gs = corr3(bfs, bfs, FFT_LEN);
283

/* start processing */
285 _enable_interrupts();

287 rxMode = txMode = 1;
rmd = rsd = rsc = 0;
289 i = 0;
while (1)
291     {
        /* if (rmd > 2*rsd && rmd > 100000)
293         mode = 2;
        else if (2*rsd > rmd && rsd > 100000)
295         mode = 1;*/

        /* tx */
297         if (txMode == 1 && uartRecvCount())
299             {
                rsd = TIM0;
            }
    }

```

```

301         txMode = 2;
302         goto RX;
303     }
304
305     if (txMode == 2 && (rsd-TIM0 >= 10000 || uartRecvCount() >= 4*BPF))
306     {
307         txMode = 3;
308         goto RX;
309     }
310
311     if (txMode == 3)
312     {
313         /* send frame for synchronization */
314         packSymbols(BURST_FR, syms);
315         cfft(syms, FFT_LEN, SCALE);
316         cbrev(syms, syms, FFT_LEN);
317         for (i = 0; i < FFT_LEN; i++)
318             syms[2*i] *= 10;
319         ifSend(syms, FFT_LEN, 2);
320
321         txMode = 4;
322         goto RX;
323     }
324
325
326     if (txMode == 4)
327     {
328         /* send frame for channel estimation */
329         packSymbols(CHAN_DET, syms);
330         sendSymbols(syms);
331
332         txMode = 5;
333         goto RX;
334     }
335
336     if (txMode == 5)
337     {
338         /* send number of bytes to expect (repeated in frame) */
339         send_count = uartRecvCount();
340         if (send_count > MAX_FR*BPF)
341             send_count = MAX_FR*BPF;
342         send_prog = 0;
343
344         for (i = 0; i < BPF/2; i++)
345             {
346                 temp[2*i] = (send_count >> 8) & 0xFF;
347                 temp[2*i] ^= otp[2*i];
348                 temp[2*i+1] = send_count & 0xFF;
349                 temp[2*i+1] ^= otp[2*i+1];
350             }
351         packBytes(temp, syms);
352         sendSymbols(syms);
353
354         txMode = 6;
355         goto RX;
356     }
357
358     if (txMode == 6)
359     {
360         /* send each frame */
361         if (send_count - send_prog < BPF)
362             {
363                 uartRecv(temp, send_count - send_prog, 1);
364                 ii = BPF;
365                 i = send_count - send_prog;
366                 //for (i = send_count-send_prog; i < BPF; i++)
367                 while (1) /* compiler bug */
368                     {
369                         temp[i] = rand();
370                         if (i >= BPF)
371                             break;
372                         i++;
373                     }
374             }
375         else
376             uartRecv(temp, BPF, 1);
377

```

```

379     packBytes(temp, syms);
        sendSymbols(syms);
        send_prog += BPF;
381
        if (send_prog >= send_count)
383             {
384                 rsd = 0;
385                 txMode = 1;
386             }
387     goto RX;
388 }
389
RX:
391
392     /* rx */
393     if (rxMode == 1 && ifRecvCount() > 2*FFT_LEN)
394     {
395         /* detect bf */
396         while (1)
397         {
398             if (ifRecvCount() < FFT_LEN)
399                 break;
400
401             dptr = ifRecvLinearize(FFT_LEN);
402
403             /*_disable_interrupts();
404             corout = corr3(bfs, dptr, FFT_LEN);
405             /*_enable_interrupts();
406
407             /*_disable_interrupts();
408             gr = corr3(dptr, dptr, FFT_LEN);
409             /*_enable_interrupts();
410
411             corsqr = ((long)corout)*corout;
412             if (gr)
413                 {
414                     corsqr2 = (corsqr)/gr;
415                     cor = (corsqr2 << 10)/gs;
416                 }
417             else
418                 cor = 0;
419
420             if (cor > corval)
421                 {
422                     rmd = TIM0;
423                     /*ifRecvDiscard(FFT_LEN+GDI>>1);
424                     ifRecv(syms, FFT_LEN, 2);
425
426                     rxMode = 2;
427                     break;
428                 }
429             else
430                 ifRecv(syms, 10, 1);
431                 /*ifRecvDiscard(10);
432
433         }
434     goto TX;
435 }
436
437     if (rxMode == 2 && /*(rmd-TIM0 >= 10000) &&*/ ifRecvCount() > 2*(FFT_LEN+GDI))
438     {
439         /* process channel estimation frame */
440         recvSymbols(syms);
441
442         ii=N; /* compiler bug */
443         for (i = 0; i < ii; i++)
444         {
445             DATA ths, th, angle;
446             /* use the matrix
447                 [ cos ths  sin ths ] * [ cos th  -sin th ] * g
448                 [ sin ths  cos ths ] * [ -sin th  cos th ]
449             where
450                 th = atan2(re, im)
451                 ths = atan2(res, ims)
452             and
453                 g = (res^2+ims^2)^0.5/(re^2+im^2)^0.5
454             which is equivalent to

```



```

455             [ res  ims ] * [ re  -im ] / ( re^2+im^2)
456             [  ims  res ]   [ -im  re ]
457     */
458     re = syms[2*(GDI-CKS)+2*i+2*BLS];
459     im = syms[2*(GDI-CKS)+2*i+2*BLS+1];
460     res = CHAN_DET[2*i];
461     ims = CHAN_DET[2*i+1];
462
463     g = (((long)res*res+(long)ims*ims)/(((long)re*re+(long)im*im)))>>10; /*
464         Q5.10*/
465     sqrt_16(&g, &g, 1);
466     g = (((long)g) * 181)>>10; /* 1/sqrt(2^15)*2^15, result is in Q5.10
467         */
468     atan2_16(&ims,&res,&ths,1);
469     atan2_16(&im,&re,&th,1);
470     angle=(ths-th)/(256)+256;
471     chanAdjMat[i][0] = (((long)SineTable[(angle+64)&0xFF]>>5)*g)>>10;
472     chanAdjMat[i][1] = (((long)SineTable[angle&0xFF]>>5)*g)>>10;
473     chanAdjMat[i][2] = (((long)-SineTable[angle&0xFF]>>5)*g)>>10;
474     chanAdjMat[i][3] = (((long)SineTable[(angle+64)&0xFF]>>5)*g)>>10;
475     }
476
477     rxMode = 3;
478     goto TX;
479 }
480
481 if (rxMode == 3 && ifRecvCount() > 2*(FFT_LEN+GDI))
482 {
483     /* process number of bytes sent */
484     recvSymbols(syms);
485     unpackBytes(temp, syms);
486     recv_count = 0;
487     recv_prog = 0;
488     for (i = 0; i < BPF; i++)
489         temp[i] ^= otp[i];
490     for (j = 0; j < 8; j++)
491     {
492         /* majority function */
493         majCntL = 0;
494         majCntH = 0;
495         mask = (1 << j);
496         for (i = 0; i < BPF/2; i++)
497         {
498             if (temp[2*i] & mask)
499                 majCntH++;
500             if (temp[2*i+1] & mask)
501                 majCntL++;
502         }
503
504         if (majCntH >= (BPF/4+HD_AGG))
505             recv_count |= mask << 8;
506         else if (majCntH <= (BPF/4-HD_AGG))
507             ;
508         else
509         {
510             failcnt++;
511             rxMode = 1;
512             goto fail;
513         }
514         if (majCntL >= (BPF/4+HD_AGG))
515             recv_count |= mask;
516         else if (majCntL <= (BPF/4-HD_AGG))
517             ;
518         else
519         {
520             failcnt++;
521             rxMode = 1;
522             goto fail;
523         }
524     }
525     if (recv_count > 10)
526         rxMode = 1;*/
527     rxMode = 4;
528 fail:
529     goto TX;
530 }
531 }

```

```

531     if (rxMode == 4 && ifRecvCount() > 2*(FFT_LEN+GDI))
532     {
533         /* receive remaining data */
534         recvSymbols(syms);
535         unpackBytes(temp, syms);
536
537         input = far_peek(dip_led);
538         if (recv_count - recv_prog < BPF)
539             {
540                 if (input & 0x10)
541                     uartSend(temp, recv_count - recv_prog, 1);
542                 else
543                     {
544                         ii = CONS;
545                         uartSend((short*)"-----\r\n", 22, 1);
546                         for (re = 0; re < ii; re++)
547                             {
548                                 uartSend(constl[re], CONS, 1);
549                                 uartSend((short*)"r\n", 2, 1);
550                             }
551                         for (re = 0; re < ii; re++)
552                             for (im = 0; im < ii; im++)
553                                 constl[re][im] = ' ';
554                     }
555                 rmd = 0;
556                 rxMode = 1;
557             }
558         else
559             if (input & 0x10)
560                 uartSend(temp, BPF, 1);
561
562         if (!(input & 0x10))
563             {
564                 ii = N;
565                 for (i = 0; i < ii; i++)
566                     {
567                         re = syms[2*(GDI-CKS)+2*i+2*BLS];
568                         re = ((unsigned)re)/(0xFFFF/(CONS+2)); //+CONS/2;
569                         im = syms[2*(GDI-CKS)+2*i+2*BLS+1];
570                         im = ((unsigned)im)/(0xFFFF/(CONS+2)); //+CONS/2;
571                         if (re < 0)
572                             re = 0;
573                         if (im < 0)
574                             im = 0;
575                         if (re >= CONS)
576                             re = CONS-1;
577                         if (im >= CONS)
578                             im = CONS-1;
579                         constl[re][im] = 'o';
580                         /*uartSend(temp, ltoa(syms[2*(GDI-CKS)+2*i+2*BLS], (char*)temp),
581                             1);
582                         uartSend((short*)"j", 2, 1);
583                         uartSend(temp, ltoa(syms[2*(GDI-CKS)+2*i+2*BLS], (char*)temp), 1);
584                         uartSend((short*)"r\n", 2, 1);*/
585                     }
586                 }
587             }
588         recv_prog += BPF;
589         goto TX;
590     }
591
592 TX:
593     far_poke(dip_led, failcnt);
594 }
595
596 void packBytes(DATA *bytes, DATA *symbols)
597 {
598     short i;
599     /*syms[0] = syms[1] = 0;
600     for (i = N+1; i < FFT_LEN-N; i++)
601         symbols[2*i] = symbols[2*i+1] = 0;*/
602     for (i = 0; i < 2*FFT_LEN; i++)
603         symbols[i] = 0;
604
605     for (i = 0; i < N/2; i++)

```

```

607     {
        symbols[2*FFT_LEN-4*i-2*BLS] = symbols[4*i+2*BLS] = QAM16GCr[(bytes[i] >> 4) & 0xF];
        symbols[2*FFT_LEN-4*i-2*BLS+1] = -(symbols[4*i+2*BLS+1] = QAM16GCi[(bytes[i] >> 4) & 0xF])
        ;
609     symbols[2*FFT_LEN-4*i-2*BLS-2] = symbols[4*i+2*BLS+2] = QAM16GCr[bytes[i] & 0xF];
        symbols[2*FFT_LEN-4*i-2*BLS-1] = -(symbols[4*i+2*BLS+3] = QAM16GCi[bytes[i] & 0xF]);
611     }
    }
613
void packSymbols(DATA *s, DATA *symbols)
615 {
    short i;
617
    /*syms[0] = syms[1] = 0;
619     for (i = N+1; i < FFT_LEN-N-2; i++)
        symbols[2*i] = symbols[2*i+1] = 0;*/
621     for (i = 0; i < 2*FFT_LEN; i++)
        symbols[i] = 0;
623
    for (i = 0; i < N; i++)
625     {
        symbols[2*FFT_LEN-2*i-2*BLS] = symbols[2*i+2*BLS] = s[2*i];
627         symbols[2*FFT_LEN-2*i-2*BLS+1] = -(symbols[2*i+2*BLS+1] = s[2*i+1]);
        }
629 }

631 void sendSymbols(DATA *symbols)
    {
633     short i;

635     ciff_t(symbols, FFT_LEN, SCALE);
        cbrev(symbols, symbols, FFT_LEN);
637
    for (i = 0; i < FFT_LEN; i++)
639         symbols[2*i] *= 10;
        /* FIX: modulate by carrier */
641     ifSend(&symbols[2*(FFT_LEN-GDI)], GDI, 2);
        ifSend(symbols, FFT_LEN, 2);
643 }

645 void unpackBytes(DATA *bytes, DATA *symbols)
    {
647     short i, re, im, rea, ima, xidx, yidx;

649     for (i = 0; i < N/2; i++)
        {
651         re = symbols[2*(GDI-CKS)+4*i+2*BLS];
            im = symbols[2*(GDI-CKS)+4*i+2*BLS+1];
653         rea = ((long)re)*chanAdjMat[2*i][0] + ((long)im)*chanAdjMat[2*i][1];
            ima = ((long)re)*chanAdjMat[2*i][2] + ((long)im)*chanAdjMat[2*i][3];
655         symbols[2*(GDI-CKS)+4*i+2*BLS] = rea;
            symbols[2*(GDI-CKS)+4*i+2*BLS+1] = ima;
657
        /* identify symbol */
659         if (rea > XDT1)
            xidx = 0;
661         else if (rea > XDT2)
            xidx = 1;
663         else if (rea > XDT3)
            xidx = 2;
665         else
            xidx = 3;
667
        if (ima > YDT1)
669             yidx = 0;
            else if (ima > YDT2)
671                 yidx = 1;
            else if (ima > YDT3)
673                 yidx = 2;
            else
675                 yidx = 3;

677         bytes[i] = QAM16GCd[xidx*4 + yidx] << 4;

679         re = symbols[2*(GDI-CKS)+4*i+2*BLS+2];
            im = symbols[2*(GDI-CKS)+4*i+2*BLS+3];
681         rea = ((long)re)*chanAdjMat[2*i+1][0] + ((long)im)*chanAdjMat[2*i+1][1];

```

```

683     ima = ((long)re)*chanAdjMat[2*i+1][2] + ((long)im)*chanAdjMat[2*i+1][3];
symbols[2*(GDI-CKS)+4*i+2*BLS+2] = rea;
symbols[2*(GDI-CKS)+4*i+2*BLS+3] = ima;
685
/* identify symbol */
687 if (rea > XDT1)
        xidx = 0;
689 else if (rea > XDT2)
        xidx = 1;
691 else if (rea > XDT3)
        xidx = 2;
693 else
        xidx = 3;
695
if (ima > YDT1)
697     yidx = 0;
else if (ima > YDT2)
699     yidx = 1;
else if (ima > YDT3)
701     yidx = 2;
else
703     yidx = 3;
705 bytes[i] |= QAM16GCd[xidx*4 + yidx];
}
707 }

709 void recvSymbols(DATA *symbols)
{
711 short i;
for (i = 0; i < GDI+FFT_LEN; i++)
713     symbols[2*i+1] = 0;

715 ifRecv(symbols, GDI+FFT_LEN, 2);
/*for (i=0; i<2*(GDI+FFT_LEN); i++)
717     symbols[i] >>= 2;*/

719 for (i = 0; i < 2*(FFT_LEN+GDI); i++)
    newBuf[i] = symbols[i];
721
//_disable_interrupts();
723 //cfft(symbols+2*(GDI-CKS), FFT_LEN, SCALE);
//cbrev(symbols+2*(GDI-CKS), symbols+2*(GDI-CKS), FFT_LEN);
725 cfft32_SCALE(newBuf+2*(GDI-CKS), FFT_LEN);
cbrev32(newBuf+2*(GDI-CKS), newBuf+2*(GDI-CKS), FFT_LEN);
727 //_enable_interrupts();

729 for (i = 0; i < 2*(FFT_LEN+GDI); i++)
    symbols[i] = newBuf[i];
731 }

733 /*****
735 // Function to compute 32 bit unsigned FTV value give f and fs
737 unsigned long ComputeFTV(unsigned long f, unsigned long fs)
{
739     unsigned idx;
    unsigned long ftv;
741
    ftv = 0;
743
    for (idx = 0; idx < FTV_CNT; idx++) {
745         if (f >= fs) {
            ftv = (ftv<<1)+1;
747             f -= fs;
        }
        else ftv <<= 1;
        f <<= 1;
751     }
    if (f >= fs) ftv += 1;
753
    return (ftv);
755 }

757 /*LDATA temp2[1024];

```

```

759 DATA corr_2 (DATA *px,DATA *py,int length)
{
761   int i;
763   LDATA out = 0;

765   for (i=0 ; i<length ; i++)
   {
767     *(temp2+i) = ((long)*(px+i)) * *(py+i);
   }
769   out = 0;

771   for (i=0 ; i<length ; i++)
773   {
775     out=out + (*(temp2+i)>>8);
   }

777   return out>>16;
}
779 */

```

if.asm

```

1 ;File name: IF.asm
;File name: AIC23int_01.asm
3 ;
; EECS 452 buffered AIC23 codec support for the C5510DSK
5 ;
;
7 ; 11Oct2003 .. initial version .. K.Metzger
; 11Apr2004 .. made small/large model independent .. KM
9 ; 8Feb2005 .. move no_isr to its own file .. KM
; 24Nov2005 .. renamed fns, improved buffering .. EJW
11 ;

13     .c54cm_off                ;don't want compatible with c54
     .ARMS_on                   ;enable assembler for ARMS=1
15     .CPL_on                   ;enable assembler for CPL=1
     .mmregs                    ;enable mem mapped register names
17

19     .global _startup, _no_isr, _resetv, _c_int00
     .global Mc2R_int, Mc2X_int
21     .global _AD_flag, _DA_flag
     .global _ifRecv, _ifRecvCount, _ifSend, _ifSendCount, _ifRecvDiscard, _ifRecvLinearize
23     .global _ifRecvSetup, _ifSendSetup
     .global _ftvR, _DDSaccumR, _ftvS, _DDSaccumS
25

     .data
27
     .bss    Mc2X_buf_adr,2,1,4    ; aligned
29     .bss    Mc2X_app_buf_off,1
     .bss    Mc2X_int_buf_off,1
31     .bss    Mc2X_buf_size,1
     .bss    Mc2X_counter,1
33     .bss    Mc2X_running_flag,1

35     .bss    Mc2R_buf_adr,2,1,4    ; aligned
     .bss    Mc2R_app_buf_off,1
37     .bss    Mc2R_int_buf_off,1
     .bss    Mc2R_buf_size,1
39     .bss    Mc2R_counter,1

41     .bss    _ftvR,2,1,2
     .bss    _DDSaccumR,2,1,2
43     .bss    _ftvS,2,1,2
     .bss    _DDSaccumS,2,1,2
45

     .text
47
     .asg    0001100000000000b,my_ST0_55
49     .asg    0110100100000000b,my_ST1_55
     .asg    1001000000000000b,my_ST2_55
51     .asg    0001000000000010b,my_ST3_55 ; ROM access is enabled

```

```

53      .asg      (0xFFFA00>>1),SINE_TABLE
55 ; Setup McBSP channel 2 codec interrupt support
57 ; for now assumes setup_codec() has been called
59
61 _startup:
62     pshboth xar0
63     mov     #_resetv >> 8, ac0      ; get int vector address page
64     mov     ac0,mmap(ivpd)         ; set up DSP int address
65     mov     ac0,mmap(ivph)         ; set up host int address
66     amov    #Mc2R_int,xar0        ; set up McB port 2 rcvr addr
67     mov     xar0,dbl*((_resetv+0x60)/2)
68     amov    #Mc2X_int,xar0        ; set up McB port 2 xmtr addr
69     mov     xar0,dbl*((_resetv+0x68)/2)
70     or      #0x3000,mmap(ifr0)    ; clear Mc2 interrupt flags
71     or      #0x3000,mmap(ier0)    ; enable Mc2TX and Mc2RX interrupts
72     mov     #0,port(#0x3003)      ; start Mc transmitter running
73     popboth xar0
74     ret
75
76 ; Support for codec interrupt driven data transfers
77
78 Mc2R_int:
79     psh     mmap(st3_55)
80     psh     mmap(T0)
81     psh     mmap(T1)
82     pshboth xar0
83     pshboth xar1
84     ;pshboth ac0
85     mov     #my_ST0_55,mmap(st0_55) ; now configure the machine
86     mov     #my_ST1_55,mmap(st1_55)
87     mov     #my_ST2_55,mmap(st2_55)
88     mov     #my_ST3_55,mmap(st3_55)
89
90     ; run the DDS to get cos and sin values
91
92     ;
93     mov     dbl*(#_DDSaccumR),ac0  ; get DDS phase accumulator
94     add     dbl*(#_ftvR),ac0      ; add the frequency tuning value
95     ;
96     mov     ac0,dbl*(#_DDSaccumR) ; and update the accumulator
97     amov    #SINE_TABLE,xar0      ; ac0 now points to sine table
98     ;
99     mov     hi(ac0<<#-8),mmap(t0) ; get top 8 bits of phase accumulator
100    ;
101    and     #0x00FF,t0             ; make sure it is 8-bit value
102    ;
103    mov     *ar0(t0),ac0          ; fetch sine value
104
105    ; end of the DDS support
106
107    ;
108    amov    #0x300000,xar1
109    mov     #0x03,*ar1
110
111    amov    #Mc2R_buf_adr,xar1      ; get buffer address
112    mov     *ar1(Mc2R_counter-Mc2R_buf_adr),T0 ; get # L&R values in buffer
113    mov     *ar1(Mc2R_buf_size-Mc2R_buf_adr),T1 ; get number allowed
114    cmp     T0==T1,TC1             ; if equal full
115    bcc     R2I_LA,!TC1           ; branch if room
116    mov     port(#0x3001),T0      ; clears the receive flag
117    b       Mc2R_exit             ; and exits...samples onto the floor
118
119 R2I_LA:
120    mov     dbl(*ar1),xar0         ; get buffer address
121    add     *ar1(Mc2R_int_buf_off-Mc2R_buf_adr),ar0 ; and calculate where to place values
122    mov     port(#0x3000),T0      ; get left value
123    mov     T0,*ar0               ; and place into buffer
124    ;;; mpy     T0,ac0
125    ;;; mov     hi(ac0),T0
126    mov     port(#0x3001),T0      ; get right value...and clear flag
127    ;mov     T0,*ar0             ; place into buffer
128    add     #1,*ar1(Mc2R_counter-Mc2R_buf_adr) ; increment count of pairs present
129    mov     *ar1(Mc2R_int_buf_off-Mc2R_buf_adr),T0 ; now update offset circularly
130    add     #1,T0                 ; increment
131    cmp     T0==T1,TC1           ; see if needs to be reset to buffer start
132    bcc     R2I_LB,!TC1         ; branch if not
133    mov     #0,T0                ; reset to buffer start
134
135 R2I_LB:
136    mov     T0,*ar1(Mc2R_int_buf_off-Mc2R_buf_adr) ; and update in memory
137
138 Mc2R_exit:

```

```

129 ;                amov    #0x300000, xar1
;                mov     #0xC, *ar1
131                ; popboth ac0
                popboth xar1
133                popboth xar0
                pop     mmap(T1)
135                pop     mmap(T0)
                pop     mmap(st3_55)
137                nop
;                ; 6 nops stops remarks 99 and 100
                nop
139                nop
                nop
141                nop
                nop
143                reti

145
; Get count of samples currently in buffer
147 ; short ifRecvCount(void);
_ifRecvCount:
149                pshboth xar3                ; use it, save it
                amov    #Mc2R_buf_adr, xar3    ; get in sample buffer address address
151                mov     *ar3(Mc2R_counter-Mc2R_buf_adr), T0 ; get count of pairs in buffer
                popboth xar3
153                ret

155
; Support to fetch codec sample values
157 ; void ifRecv(short *samples, short count, short stride);
_ifRecv:
159                pshboth xar2                ; use it, save it
                pshboth xar3                ; use it, save it
161                amov    #Mc2R_buf_adr, xar3    ; get in sample buffer address address
Mc2R_wait:
163                mov     *ar3(Mc2R_counter-Mc2R_buf_adr), T2 ; get count of pairs in buffer
                sub     T0, T2
165                bcc    Mc2R_wait, T2<#0      ; wait if there aren't any
                mov     dbl(*ar3), xar2        ; set up buffer address
167                ; ; ; ; ; add    dbl(*ar3), ar2
                sub     #1, T0
169                mov     T0, brc0
                mov     *ar3(Mc2R_app_buf_off-Mc2R_buf_adr), T0
171                mov     *ar3(Mc2R_buf_size-Mc2R_buf_adr), T2 ; get limiting value
                rptb    RCOPY_LOOP-1
173                mov     *ar2(T0), *(ar0+T1)    ; and place in caller's location
;                mov     *ar2, T0                ; fetch R value
175 ;                mov     T0, *ar1            ; and place in caller's location
                sub     #1, *ar3(Mc2R_counter-Mc2R_buf_adr) ; indivisible decrement of count
177                add     #1, T0                ; increment
                cmp     T0==T2, TC1            ; if equal need to reset to 0
179                bcc    R2_LA, !TC1           ; branch if not equal
                mov     #0, T0                ; zero to start of buffer
181 R2_LA:
                nop
183 RCOPY_LOOP:
                mov     T0, *ar3(Mc2R_app_buf_off-Mc2R_buf_adr) ; and update in memory
185                popboth xar3
                popboth xar2
187                ret

189

191 ; short *ifRecvLinearize(short count);
_ifRecvLinearize:
193                pshboth xar2                ; use it, save it
                pshboth xar3                ; use it, save it
195                amov    #Mc2R_buf_adr, xar3    ; get in sample buffer address address
RL_wait:
197                mov     *ar3(Mc2R_counter-Mc2R_buf_adr), T2 ; get count of pairs in buffer
                sub     T0, T2
199                bcc    RL_wait, T2<#0        ; wait if there aren't any
                mov     dbl(*ar3), xar2        ; set up buffer address
201
                mov     *ar3(Mc2R_buf_size-Mc2R_buf_adr), T1 ; get limiting value
203                sub     *ar3(Mc2R_app_buf_off-Mc2R_buf_adr), T1
205                cmp     T1>=T0, TC1

```

```

207         bcc          RL_END,TC1
209         sub          T1,T0
211         mov          *ar3(Mc2R_buf_size-Mc2R_buf_adr),T1      ; get limiting value
MOV        dbl(*ar3),xar2
213         MOV        xar2,xar3
add        T1,ar2
215         sub          #1,T0
MOV        T0,BRC0
217         RPTB        RL_END-1
MOV        *ar3+,*ar2+
219 RL_END:
amov      #Mc2R_buf_adr,xar3      ; get in sample buffer address address
221         amov      #Mc2R_buf_adr,xar0
mov        dbl(*ar0),xar0
223         add        *ar3(Mc2R_app_buf_off-Mc2R_buf_adr),ar0
popboth   xar3
225         popboth   xar2
ret
227
229 ; ifRecvSetup(*buffer, size);
_ifRecvSetup:
231         amov      #Mc2R_buf_adr,xar1
mov        xar0,dbl(*ar1)      ; get the A/D in buffer address address
233         mov        T0,*ar1(Mc2R_buf_size-Mc2R_buf_adr)      ; get the L&R pair count
mov        #0,*ar1(Mc2R_app_buf_off-Mc2R_buf_adr) ; initialize application level buffer
235         mov        offset #0,*ar1(Mc2R_int_buf_off-Mc2R_buf_adr) ; initialize interrupt level buffer
mov        offset #0,*ar1(Mc2R_counter-Mc2R_buf_adr)      ; nothing present yet
237         ret
239
; Support to fetch codec sample values
241 ; void ifRecvDiscard(short count);
_ifRecvDiscard:
243         pshboth   xar2          ; use it, save it
pshboth   xar3          ; use it, save it
245         amov      #Mc2R_buf_adr,xar3      ; get in sample buffer address address
IFRD_WAIT:
247         mov        *ar3(Mc2R_counter-Mc2R_buf_adr),T2 ; get count of pairs in buffer
sub        T0,T2
249         bcc        IFRD_WAIT,T2<#0      ; wait if there aren't any
mov        dbl(*ar3),xar2      ; set up buffer address
251         mov        *ar3(Mc2R_app_buf_off-Mc2R_buf_adr),T1 ; now update offset circularly
add        T1,T0              ; increment
253         mov        *ar3(Mc2R_buf_size-Mc2R_buf_adr),T2      ; get limiting value
cmp        T1>=T2,TC1          ; if equal need to reset to 0
255         bcc        IFRD_NOP,!TC1        ; branch if not equal
sub        T2,T1
257 IFRD_NOP:
mov        T1,*ar3(Mc2R_app_buf_off-Mc2R_buf_adr) ; and update in memory
259         mov        *ar3(Mc2R_counter-Mc2R_buf_adr),T1
sub        T0,T1
261         mov        T0,*ar3(Mc2R_counter-Mc2R_buf_adr)      ; indivisible decrement of count
263         popboth   xar3
popboth   xar2
265         ret
267
269 ;-----
;
271 ; Support to send L&R sample values to the AIC23 codec
273 Mc2X_int:
psh        mmap(st3_55)
275         psh        mmap(T0)
psh        mmap(T1)
277         pshboth   xar0
pshboth   xar1
279         ;pshboth   ac0
mov        #my_ST0_55,mmap(st0_55) ; now configure the machine

```



```

281         mov     #my_ST1_55,mmap(st1_55)
           mov     #my_ST2_55,mmap(st2_55)
283         mov     #my_ST3_55,mmap(st3_55)

285         ; run the DDS to get cos and sin values

287         ;mov     dbl*(#_DDSaccumS),ac0 ; get DDS phase accumulator
           ;add     dbl*(#_ftvS),ac0 ; add the frequency tuning value
289         ;mov     ac0,dbl*(#_DDSaccumS) ; and update the accumulator
           ;amov    #SINE_TABLE,xar0 ; ac0 now points to sine table
291         ;mov     hi(ac0<<#-8),mmap(t0) ; get top 8 bits of phase accumulator
           ;and     #0x00FF,t0 ; make sure it is 8-bit value
293         ;mov     *ar0(t0),ac0 ; fetch sine value

295         ; end of the DDS support

297         amov    #Mc2X_buf_adr,xar1 ; get TX buffer address address
           mov     dbl(*ar1),xar0 ; get TX buffer address
299         mov     *ar1(Mc2X_counter-Mc2X_buf_adr),T0 ; get count of pairs present in buffer
           bcc     X2I_LA,T0==0 ; if none nothing to do
301         sub     #1,*ar1(Mc2X_counter-Mc2X_buf_adr) ; we will send a LR pair reducing the count
           add     *ar1(Mc2X_int_buf_off-Mc2X_buf_adr),ar0 ; and add in offset
303         mov     *ar0,T0 ; get L value from buffer
           ;mpy    T0,ac0
305         ;mov     hi(ac0),T0
           mov     T0,port(#0x3002) ; and send to L in TX
307         mov     #0,port(#0x3003) ; and send to R in TX and clear flag
           mov     #1,*ar1(Mc2X_running_flag-Mc2X_buf_adr) ; note we expecting an interrupt
309         mov     *ar1(Mc2X_int_buf_off-Mc2X_buf_adr),T0 ; now need to up interrupt buffer
           offset
           add     #1,T0 ; circularly
311         mov     *ar1(Mc2X_buf_size-Mc2X_buf_adr),T1 ; compare offset with buffer size
           cmp     T0==T1,TC1 ; if equal need to reset to 0
313         bcc     X2I_LB,!TC1 ; branch if not needed to reset to 0
           mov     #0,T0 ; get the zero
315 X2I_LB:
           mov     T0,*ar1(Mc2X_int_buf_off-Mc2X_buf_adr) ; and update the value in memory
317         b       X2I_exit ; all done so exit
319 X2I_LA:
           mov     #0,*ar1(Mc2X_running_flag-Mc2X_buf_adr) ; note we are not expecting an
           interrupt
321         mov     #0,port(#0x3002) ;
           mov     #0,port(#0x3003) ;
323 X2I_exit:
           ;popboth ac0
           popboth xar1
325         popboth xar0
           pop     mmap(T1)
327         pop     mmap(T0)
           pop     mmap(st3_55)
329         nop ; 6 nops stop remarks 99 and 100
           nop
331         nop
333         nop
           nop
335         reti

337
; Get count of unused sample spots currently in buffer
339 ; short ifSendCount(void);
_ifSendCount:
341         pshboth xar3 ; use it, save it
           amov    #Mc2X_buf_adr,xar3 ; get in sample buffer address address
343         mov     *ar3(Mc2X_buf_size-Mc2X_buf_adr),T0 ; number of spaces available
           sub     *ar3(Mc2X_counter-Mc2X_buf_adr),T0 ; number values present
345         popboth xar3
           ret

347
; Application level function to send samples to DAC
349 ; void ifSend(short *samples, short count, short stride);
_ifSend:
351 XL2_LC:
           amov    #Mc2X_buf_adr,xar1 ; point to buffer address address
353         mov     *ar1(Mc2X_buf_size-Mc2X_buf_adr),T2 ; number of spaces available
           sub     *ar1(Mc2X_counter-Mc2X_buf_adr),T2 ; number values present
355         cmp     T2<=T0,TC1 ; see if they are equal

```

```

        bcc     XL2_LC,TC1                ; wait if no room
357      mov     dbl(*ar1),xar2            ; get buffer address
        ;;;; add     *ar1(Mc2X_app_buf_off-Mc2X_buf_adr),ar2
359      psh     T0
        sub     #1,T0
361      mov     T0,brc0
        mov     *ar1(Mc2X_buf_size-Mc2X_buf_adr),T2 ; number of spaces available
363      mov     *ar1(Mc2X_app_buf_off-Mc2X_buf_adr),T0
        ;;;; add     #1,T0                ; increment
365      rptb    SCOPY_LOOP-1
        mov     *(ar0+T1),*ar2(T0)       ; store left value into buffer
367      add     #1,T0                    ; increment
        cmp     T0==T2,TC1               ; may need to reset
369      bcc     X2_LB,!TC1              ; not yet
        mov     #0,T0                    ; put back to buffer start
371 X2_LB:
        nop
373 SCOPY_LOOP:
        bset    intm                      ; disable interrupts
375      mov     T0,*ar1(Mc2X_app_buf_off-Mc2X_buf_adr) ; update the putting offset
        pop     T0
377      mov     *ar1(Mc2X_counter-Mc2X_buf_adr),T1
        add     T0,T1
379      mov     T1,*ar1(Mc2X_counter-Mc2X_buf_adr) ; increment count
        mov     *ar1(Mc2X_running_flag-Mc2X_buf_adr),T0
381      bcc     X2_LA,T0!=0              ; branch if xmtr running
        intr    #0xD                      ; trigger the interrupt if not
383 X2_LA:
        bclr    intm                      ; reenale interrupts
385 X2_exit:
        ret
387
389 ; ifSendSetup(*buffer, size);
391 _ifSendSetup:
        amov    #Mc2X_buf_adr,xar1        ; point to buffer address address
393      mov     xar0,dbl(*ar1)            ; save address of L&R output buffer
        mov     T0,*ar1(Mc2X_buf_size-Mc2X_buf_adr) ; save number of L&R pairs
395      mov     #0,*ar1(Mc2X_app_buf_off-Mc2X_buf_adr) ; initialize application level offset
        value
        mov     #0,*ar1(Mc2X_int_buf_off-Mc2X_buf_adr) ; initialize interrupt level offset
        value
397      mov     #0,*ar1(Mc2X_counter-Mc2X_buf_adr) ; nothing in the buffer yet
        mov     #0,*ar1(Mc2X_running_flag-Mc2X_buf_adr) ; and the TX is not going to interrupt
        us     yet
399      ret

```

uart.c

```

1 /* File name: UART.c
   File name: UART2support.c
3
   Buffered interrupt support for DSP Global UART board
5   on the TI C5510 DSK.
7   UART channel 2 only.
9   28Mar2004 .. initial version .. KM
   29Mar2004 .. UART 2 int sup evolved from test code .. KM
11  24Nov2005 .. renamed fns, improved buffering .. EJW
13 */
15 #define FOREVER 1
17 #define UART 0x500200
   #define RBR (UART+0x00)
19 #define THR (UART+0x00)
   #define DLL (UART+0x00)
21 #define DLM (UART+0x02)
   #define IER (UART+0x02)
23 #define ISR (UART+0x04)
   #define FCR (UART+0x04)
25 #define LCR (UART+0x06)

```

```

#define MCR (UART+0x08)
27 #define LSR (UART+0x0A)
#define MSR (UART+0x0C)
29 #define SPR (UART+0x0E)

31
#define IER0 ((unsigned long)0x00)
33 #define IFR0 ((unsigned long)0x01)
#define IVPD ((unsigned long)0x49)
35 #define IVPH ((unsigned long)0x4A)

37 #define INT0 0x0008
#define INT0_BIT 0x0004
39
interrupt void UART2int(void);
41 void resetv();

43 volatile int *U2RxAdr, U2RxSize, U2RxAOff, U2RxIOff, U2RxCount;
volatile int *U2TxAdr, U2TxSize, U2TxAOff, U2TxIOff, U2TxCount, U2TxStopped;
45
#define far_poke FarPoke
47 #define far_peek FarPeek

49 //-----

51
/* Function to set up the UART channel buffered Rx and Tx support.
53
Call prior to globally enabling the interrupt system.
55
The arguments are:
57
*inbuf    A pointer to the receive (Rx) buffer.
59 nin      The number of ints in the Rx buffer.
*outbuf    A pointer to the transmit (Tx) buffer.
61 nout     The number of ints in the Tx buffer.
RateDiv    The rate divisor value baud rate is
63          230,400/RateDiv. For 38400 baud use a
          value of 6.
65 */

67 void uartSetup(int *inbuf, int nin, int *outbuf, int nout, int RateDiv)
{
69     unsigned long resetloc;

71     // set up buffering support

73     U2RxAdr = inbuf;    // address of Rx buffer
U2RxSize = nin;        // size of Rx buffer
75     U2RxAOff = 0;      // application level Rx address offset
U2RxIOff = 0;          // interrupt level Rx address offset
77     U2RxCount = 0;     // count of characters in Rx buffer
U2TxAdr = outbuf;     // address of Tx buffer
79     U2TxSize = nout;   // size of Tx buffer
U2TxAOff = 0;          // application level TX address offset
81     U2TxIOff = 0;     // interrupt level Tx address offset
U2TxCount = 0;        // count of characters in Tx buffer
83     U2TxStopped = 1;  // Tx waiting for a character to be sent

85     // set up interrupt vector and interrupt registers

87     resetloc = (long)resetv;
far_poke(IVPD, (unsigned)(resetloc>>8));
89     far_poke(IVPH, (unsigned)(resetloc>>8));
far_poke((resetloc>>1)+INT0, (unsigned)((unsigned long)UART2int>>16));
91     far_poke((resetloc>>1)+INT0+1, (unsigned)((unsigned long)UART2int));
far_poke(IER0, far_peek(IER0)|INT0_BIT);
93     far_poke(IFR0, INT0_BIT);

95     while (!(far_peek(ISR) & 0x1))
        ;

97
#if 1
99     // configure the UART channel 2

101    far_poke(LCR, 0x80); // access baud rate registers
far_poke(DLM, RateDiv>>8); // set baud rate divisor high byte

```

```

103     far_poke(DLL, RateDiv);           // set baud rate division low byte
105 //     far_poke(LCR, 0xBF);           // access enhanced registers
106 //     far_poke(ISR, 0xC0);
107     far_poke(LCR, 0x03);           // use 8 data and 1 stop bit
109     far_poke(FCR, 0x0F);           // insure FIFOs are on
110     far_poke(LSR, 0x60);           // initialize line status register
111     far_poke(IER, 0x0F);           // have UART generate Rx and Tx interrupts
112 //     far_poke(IER, 0x07);           // have UART generate Rx and Tx interrupts
113     far_poke(MCR, 0x0B);           // make UART int req outputs active, set flow control
114 //     far_poke(MCR, 0x08);           // make UART int req outputs active
115     far_peek(RBR);                 // clear receiver buffer
116 #else
117 // configure the UART channel 2
119     far_poke(LCR, 0x80); // access baud rate registers
120     far_poke(DLM, RateDiv>>8); // set baud rate divisor high byte
121     far_poke(DLL, RateDiv); // set baud rate division low byte
122     far_poke(LCR, 0x07); // use 8 data and 2 stop bits
123
124     far_poke(FCR, 0x00); // insure FIFOs are off
125     far_poke(LSR, 0x60); // initialize line status register
126     far_poke(IER, 0x03); // have UART generate Rx and Tx interrupts
127     far_poke(MCR, 0x08); // make UART int req outputs active
128     far_peek(RBR); // clear receiver buffer
129 #endif
131     return;
132 }
133 }
134 //
135 //
136 int uartRecvCount()
137 {
138     return U2RxCount;
139 }
140 // Function to fetch characters from the Rx buffer
141 void uartRecv(short *buf, short count, short stride)
142 {
143     int i;
144     short *ptr = buf;
145     while (U2RxCount < count); // wait if not enough characters present
146     for (i = 0; i < count; i++)
147     {
148         *ptr = *(U2RxAdr+U2RxAOff); // fetch character
149         if (++U2RxAOff >= U2RxSize) U2RxAOff = 0; // adv pointer cyclicly
150         ptr += stride;
151     }
152     _disable_interrupts(); // enter a critical section
153     U2RxCount -= count; // reduce the number present
154     _enable_interrupts(); // exit the critical section
155     if (U2RxCount < U2RxSize-400)
156         far_poke(MCR, far_peek(MCR)|0x02);
157 }
158 short uartSendCount()
159 {
160     return (U2TxSize - U2TxCount);
161 }
162 // Function to put characters into the Tx buffer
163 void uartSend(short *buf, short count, short stride)
164 {
165     int i;
166     short *ptr = buf;
167     while (U2TxCount >= U2TxSize); // wait if Tx buffer full
168     for (i = 0; i < count; i++)
169     {
170         *ptr = *(U2TxAdr+U2TxAOff); // fetch character
171         if (++U2TxAOff >= U2TxSize) U2TxAOff = 0; // adv pointer cyclicly
172         ptr += stride;
173     }
174     _disable_interrupts(); // enter a critical section
175     U2TxCount += count; // increase the number present
176     _enable_interrupts(); // exit the critical section
177 }

```

```

181     while ((U2TxSize-U2TxCount) < count)
182         i; // wait if no room in the buffer
183     _disable_interrupts(); // enter a critical section
184
185     for (i = 0; i < count; i++)
186     {
187         if (U2TxStopped != 0) { // if Tx not running
188             far_poke(THR, *ptr); // load character directly
189             far_poke(IER, far_peek(IER)|0x0002); // reenale interrupt
190             U2TxStopped = 0; // and note that is expected
191         }
192         else { // if Tx is running
193             *(U2TxAdr+U2TxAOFF) = *ptr; // put character into buffer
194             if (++U2TxAOFF >= U2TxSize) U2TxAOFF = 0; // adv pointer cyclicly
195             U2TxCount++; // and increase the count
196         }
197         ptr += stride;
198     }
199     _enable_interrupts(); // exit the critical section
200
201     return;
202 }
203
204 //-----
205
206 // Interrupt handler for UART channel 2
207
208 int volatile U2Flag;
209
210 interrupt void UART2int(void)
211 {
212     while (1)
213     {
214         U2Flag = far_peek(ISR); // get the interrupt status value
215         if ((U2Flag&0x0C) != 0) { // true if Rx interrupt
216             while (far_peek(LSR) & 0x1)
217             {
218                 if (U2RxCount < U2RxSize-200) { // ignore if no room
219                     *(U2RxAdr+U2RxIOFF) = far_peek(RBR); // put character in Rx buffer
220                     if (++U2RxIOFF >= U2RxSize) U2RxIOFF = 0; // adv pointer cyclicly
221                     U2RxCount++; // count the character
222                 }
223                 else { // if no room in the Rx buffer we
224                     // far_peek(RBR); // fetch the character and discard it
225                     far_poke(MCR, far_peek(MCR)&(~0x02));
226                     far_poke(MCR, 0x08);
227                     far_poke(MSR, 0x90);
228                     U2Flag = far_peek(MCR);
229                     U2Flag = far_peek(MSR);
230                     far_peek(RBR); // fetch the character and discard it
231                     *(U2RxAdr+U2RxIOFF) = far_peek(RBR); // put character in Rx buffer
232                     if (++U2RxIOFF >= U2RxSize) U2RxIOFF = 0; // adv pointer cyclicly
233                     U2RxCount++; // count the character
234                 }
235             }
236             //far_peek(RBR);
237         }
238     }
239     else if ((U2Flag&0x02)!=0) { // true if Tx interrupt
240         if (U2TxCount == 0) { // true if no characters in Tx buffer
241             U2TxStopped = 1; // so note that no future interrupt
242             //far_poke(IER, far_peek(IER)&0xFFFD); // and disable Tx interrupt requests
243         }
244         else { // otherwise have a character to send
245             if (!U2TxStopped)
246             /*U2Flag = far_peek(MSR);
247             if (!(U2Flag & 0x10) || !(U2Flag & 0x20))
248             {
249                 short i = 0;
250             }
251             else*/
252             {
253                 far_poke(THR, *(U2TxAdr+U2TxIOFF)); // so put into the Tx buffer
254                 if (++U2TxIOFF >= U2TxSize) U2TxIOFF = 0; // adv pointer cyclicly
255                 U2TxCount--; // reduce count present

```

```

257     }
258   }
259   }
260   else if (U2Flag&0x1)
261   {
262     break;
263     //unsigned short i = far_peek(IER);
264   }
265   else if (!(U2Flag&0x3F))
266   {
267     /* MSR change */
268     U2Flag = far_peek(MSR);
269     /*if (U2Flag & 0x2)
270     {
271       if (U2TxStopped)
272         U2TxStopped = 0;
273       else
274         U2TxStopped = 1;
275       //far_poke(IER, far_peek(IER)&0xFFFD);
276     }*/
277   }
278   else
279   {
280     short i = 0;
281   }
282   //else {
283   //  while(0); // should never get here..but if we do, wait for help
284   //}
285   return;
286 }
287 unsigned short iiii;
288 void ChErrors()
289 {
290 unsigned long resetloc;
291
292
293 resetloc = (long)resetv;
294
295 iiii = far_peek(LSR);
296 if (iiii & 0xE)
297 {
298     iiii=far_peek(RBR);
299 }
300 iiii = far_peek(IER);
301 if (~iiii & 0x03)
302     iiii=0;
303
304
305
306
307     if (far_peek(IVPD) != (unsigned)(resetloc >>8))
308         iiii=far_peek(IVPD);
309     if (far_peek(IVPH) != (unsigned)(resetloc >>8))
310         iiii=far_peek(IVPH);
311     if (far_peek((resetloc >>1)+INT0) != (unsigned)((unsigned long)UART2int >>16))
312         iiii=far_peek((resetloc >>1)+INT0);
313     if (far_peek((resetloc >>1)+INT0+1) != (unsigned)((unsigned long)UART2int&0xFFFF))
314         iiii=far_peek((resetloc >>1)+INT0+1);
315     iiii = far_peek(IER0); //, far_peek(IER0)|INT0_BIT);
316     if (far_peek(IFR0) != INT0_BIT)
317         iiii=far_peek(IFR0);
318
319 }

```

corr.asm

```

1 ;*****
2 ; Version 2.31.00
3 ;*****
4 ; Function: convol
5 ; Processor: C55xx
6 ; Description: Implements real convolution algorithm using
7 ;               single-MAC approach. C-callable.
8 ;
9 ; Usage: ushort oflag = firs(DATA *x,

```

```

;
;          DATA *h,
11 ;          DATA *r,
;          ushort nr,
13 ;          ushort nh)
;
15 ; Copyright Texas instruments Inc, 2000
;*****
17
19     .ARMS_off           ;enable assembler for ARMS=0
     .CPL_on             ;enable assembler for CPL=1
     .mmregs             ;enable mem mapped register names
21
; Stack frame
23 ;
RET_ADDR_SZ     .set 1           ;return address
25 REG_SAVE_SZ   .set 0           ;save-on-entry registers saved
FRAME_SZ       .set 0           ;local variables
27 ARG_BLK_SZ   .set 0           ;argument block

29 PARAM_OFFSET .set ARG_BLK_SZ + FRAME_SZ + REG_SAVE_SZ + RET_ADDR_SZ

31
; Register usage
33 ;
     .asg     AR0, x_ptr         ;linear pointer
35     .asg     AR1, h_ptr       ;circular pointer
;;;     .asg     AR2, r_ptr       ;linear pointer
37
;;;     .asg     BSA01, h_base    ;base addr for h_ptr
39 ;;;     .asg     BK03, h_sz     ;circ buffer size for h_sz

41 ;;;     .asg     BRC0, inner_cnt ;inner loop count
     .asg     CSR, inner_cnt     ;inner loop count
43 ;;;     .asg     BRC0, outer_cnt ;outer loop count

45 ;;;     .asg     T0, oflag      ;returned value

47 ;;;ST2mask .set 000000000000010b ;circular/linear pointers

49
     .global _corr3
51
     .text
53 _corr3:

55 ;
; Allocate the local frame and argument block
57 ;-----
;     SP = SP - #(ARG_BLK_SZ + FRAME_SZ + REG_SAVE_SZ)
59 ; - not necessary for this function (the above is zero)

61 ;
; Save any save-on-entry registers that are used
63 ;-----
     PSH     mmap(ST0_55)
65     PSH     mmap(ST1_55)
     PSH     mmap(ST2_55)
67     PSH     mmap(ST3_55)

69 ;
; Configure the status registers as needed.
71 ;-----
73 ;     AND     #001FFh, mmap(ST0_55) ;clear all ACOVx,TC1, TC2, C
;
75 ;     OR      #04140h, mmap(ST1_55) ;set CPL, SXMD, FRCT;
;
77 ;     AND     #0F9DFh, mmap(ST1_55) ;clear M40, SATD, 54CM
;
79 ;     AND     #07A00h, mmap(ST2_55) ;clear ARMS, RDM, CDPLC, AR[0-7]LC;
;
81 ;     AND     #0FFDDh, mmap(ST3_55) ;clear SATA, SMUL

83 ;BCLR INTM

85     .asg     0001100000000000b,my_ST0_55
     .asg     0110100100000000b,my_ST1_55

```

```

87      .asg      1001000000000000b,my_ST2_55
88      .asg      00010000000000010b,my_ST3_55 ; ROM access is enabled
89
90      mov      #my_ST0_55,mmap(st0_55) ; now configure the machine
91      mov      #my_ST1_55,mmap(st1_55)
92      mov      #my_ST2_55,mmap(st2_55)
93      mov      #my_ST3_55,mmap(st3_55)
94      BSET M40
95      BCLR FRCT
96      BCLR SATD
97      BCLR CARRY
98      BCLR SATA
99      BCLR SMUL
100     BSET SXMD
101
102     NOP
103     NOP
104     NOP
105     NOP
106     NOP
107     NOP
108     NOP
109     NOP
110     NOP
111     NOP
112     NOP
113     NOP
114     NOP
115     NOP
116     NOP
117     NOP
118     ;
119     ; Setup passed parameters in their destination registers
120     ; Setup circular/linear CDP/ARx behavior
121     ;-----
122
123     ; x pointer - passed in its destination register , need do nothing
124
125     ; h pointer - setup
126
127     ;;;;MOV      mmap(AR1), h_base      ;base address of h[]
128
129     ;;;SUB #1, T1, h_ptr      ;h_ptr = nh-1 (end of h[])
130     ;;;mov #0, h_ptr
131
132     ;;;;;;MOV      mmap(T1), h_sz      ;coefficient array size
133
134     ; r pointer - passed in its destination register , need do nothing
135
136     ; Set circular/linear ARx behavior
137
138     ;;;;;;;;;;;;;;MOV #ST2mask, mmap(ST2_55) ;configure circular/linear pointers
139
140     ;
141     ; Setup loop counts
142     ;-----
143     SUB #1, T0      ;T0 = nr - 1
144     MOV T0, outer_cnt ;outer loop executes nr times
145     SUB #3, T1, T0 ;T0 = nh - 3
146     MOV T0, inner_cnt ;inner loop executes nh-2 times
147
148     ;
149     ; Compute last iteration input pointer offsets
150     ;-----
151     SUB #2, T1      ;T1 = nh-2, adjustment for x_ptr
152
153     ;
154     ; Start of outer loop
155     ;-----
156     ; ||RPTBLOCAL loop1-1 ;start the outer loop
157
158     ;1st iteration
159     mov #0,AC0
160     ;;;;mpy      ac0,ac0
161     ;;;MPYM *x_ptr+, *h_ptr+, AC0
162     ;;;SFTS AC0,#-8
163

```



```

165 ;inner loop
;      ||RPT    inner_cnt
167      RPT     inner_cnt
      MACM     *x_ptr+, *h_ptr+, AC0
169
;;      BCLR ACOV0
171 ;;      BCLR ACOV1
;;      RPTBLOCAL looppp-1
173
;      mov     *x_ptr+,AC1
175      SFTS   ac1,#-4
;      mov     ac1,T1
177      mov     *h_ptr+,AC1
;;      SFTS   ac1,#-4
179      SFTS   T0,#-4
;      MPYM40  mmap(T1),AC1, AC1
181 ;;;      MPYM40  *x_ptr+,*h_ptr+, AC2
;      MPYM     *x_ptr+, *h_ptr+, AC1
183      ;;;SFTS   AC2,#-8
;;;      add     AC2,AC0
185 ;;;      MOV     AC2,AC3
;;;      abs     AC2
187 ;;;      SFTS   AC2,#-24
;;;      AND     #256,AC2
189 ;;;      mov     AC2,T0
;;;      cmp     mmap(T0)==#0,TC1
191 ;;;      BCC   ASDF2,TC1
;;;      NOP
193 ;;;ASDF2:
;;;      NOP
195 ;;;looppp:

197 ;last iteration has different pointer adjustment and rounding
      ;;;MACMR  *(x_ptr-T1), *h_ptr+, AC0
199
;;;      mov     AC0,AC1
201      SFTS   AC0,#-7
;store result to memory
203      ROUND  AC0
      MOV     hi(AC0), T0      ;store Q15 result to memory
205 ;;;      abs     T0,T1
;;;      AND     #256,T1,T1
207 ;;;      CMP     mmap(T1)==#0,TC1
;;;      BCC   ASDF,TC1
209 ;;;      NOP
;;;ASDF:
211
;;;loop1:                                ;end of outer loop
213
;
215 ; Check if overflow occurred, and setup return value
-----
217 ;      MOV     #0, oflag                ;clear oflag
219 ;      XCCPART check1, overflow(AC0)    ;clears ACOV0
;      ||MOV   #1, oflag                ;overflow occurred
221 ;check1:
223 ;
; Restore status regs to expected C-convention values as needed
225 -----
227      BCLR   FRCT                        ;clear FRCT
      ;;;AND   #0FE00h, mmap(ST2_55)    ;clear CDPLC and AR[7-0]LC
229      BSET   ARMS                        ;set ARMS
231
;
233 ; Restore any save-on-entry registers that are used
-----
235      POP    mmap(ST3_55)
237      POP    mmap(ST2_55)
      POP    mmap(ST1_55)
      POP    mmap(ST0_55)
239
;BSET INTM

```

```

241 ;
243 ; Deallocate the local frame and argument block
;-----
245 ;     SP = SP + #(ARG_BLK_SZ + FRAME_SZ + REG_SAVE_SZ)
; - not necessary for this function (the above is zero)
247 ;
;
249 ; Return to calling function
;-----
251     RET                                ;return to calling function
;-----
253 ;
;End of file

```

ofdm.cmd

```

/*****
2  LINKER command file for EECS 452 C5510DSK memory map.
   Small memory model --- Version 1.0 25Jul2003 KM
4  Added large pages --- Version 1.01 18Nov2003 KM

6  Appears to work ok for large memory model as well.
   Linker represents addresses and allocations using 8-bit bytes!!!!!!
8
   *****/
10
12 -stack 0x2000 /* Primary stack size .. fills one 8KB block */
14 -sysstack 0x1000 /* Secondary stack size .. fills one half 8KB block */
16 -heap 0x2000 /* Heap area size .. fills one 8KB block */

18 -c /* Use C linking conventions: auto-init vars at runtime */
20 -u _Reset /* Force load of reset interrupt handler */

22 MEMORY
{
24 PAGE 0: /* --- Unified Program/Data Address Space --- */
   MMR_RSVD : origin = 0x000000, length = 0x0000BF /* 192 bytes MMR reserved */
26   VECT (RWIX) : origin = 0x000100, length = 0x000100 /* 256 byte interrupt vector */
   DARAM (RWIX) : origin = 0x000200, length = 0x00FD00 /* almost 64KB of DARAM */
28   SARAM0 (RWIX) : origin = 0x010000, length = 0x010000 /* 64KB of SARAM */
   SARAM1 (RWIX) : origin = 0x020000, length = 0x020000 /* 128KB of SARAM */
30   SARAM2 (RWIX) : origin = 0x040000, length = 0x010000 /* 64KB of SARAM */
   /* SDRAM has 0xB0000 37776 KB of SDRAM .. notall allocated here */
32   SDRAM0 (RWIX) : origin = 0x050000, length = 0x010000 /* 64KB of SDRAM */
   SDRAM1 (RWIX) : origin = 0x060000, length = 0x020000 /* 128KB of SDRAM */
34   SDRAM2 (RWIX) : origin = 0x080000, length = 0x020000 /* 128KB of SDRAM */
   SDRAM3 (RWIX) : origin = 0x0A0000, length = 0x020000 /* 128KB of SDRAM */
36   SDRAM4 (RWIX) : origin = 0x0C0000, length = 0x020000 /* 128KB of SDRAM */
   SDRAM5 (RWIX) : origin = 0x0E0000, length = 0x020000 /* 128KB of SDRAM */
38   SDRAM6 (RWIX) : origin = 0x100000, length = 0x020000 /* 128KB of SDRAM */
   SDRAM7 (RWIX) : origin = 0x120000, length = 0x020000 /* 128KB of SDRAM */
40   SDRAM8 (RWIX) : origin = 0x140000, length = 0x020000 /* 128KB of SDRAM */
   SDRAM9 (RWIX) : origin = 0x160000, length = 0x020000 /* 128KB of SDRAM */
42   SDRAM10 (RWIX) : origin = 0x180000, length = 0x020000 /* 128KB of SDRAM */
   SDRAM11 (RWIX) : origin = 0x1A0000, length = 0x020000 /* 128KB of SDRAM */
   FLASH : origin = 0x400000, length = 0x80000
44   VECS (RIX) : origin = 0xffff00, length = 0x000100 /* 256-byte int vector*/
}

46 SECTIONS
{
   .text > SARAM0 PAGE 0 /* Code */

48 /* These sections must be on same physical memory page */
/* when small memory model is used */
50
   .data > DARAM PAGE 0 /* Initialized vars */
52   .bss > DARAM PAGE 0 /* Global & static vars */
   .const > DARAM PAGE 0 /* Constant data */
54   .system > DARAM PAGE 0 /* Dynamic memory (malloc) */
   .stack > DARAM PAGE 0 ALIGN = 0x2000 /* Primary system stack */
56   .sysstack > DARAM PAGE 0 ALIGN = 0x2000 /* Secondary system stack */
   .cio > SARAM0 PAGE 0 /* C I/O buffers */
58

```

```

        .fftcode    > DARAM PAGE 0
60
/* These sections may be on any physical memory page */
62 /* when small memory model is used */

64 .switch        > SARAM0 PAGE 0 /* Switch statement tables */
   .cinit         > SARAM0 PAGE 0 /* Auto-initialization tables */
66 .pinit         > SARAM0 PAGE 0 /* Initialization fn tables */

68   vectors      > VECT  PAGE 0 /* Interrupt vectors */

70   buffers              > SDRAM1 PAGE 0
   buffers2             > SDRAM2 PAGE 0
72   aicinput           > SDRAM3 PAGE 0

74 /* Allocate pages in SARAM for when using large memory model */

76 /*   SARAMA        > SARAM0 PAGE 0
   SARAMB          > SARAM1 PAGE 0
78   SARAMC          > SARAM2 PAGE 0 */

80 /* Allocate pages in SDRAM for when using large memory model */

82 /*   SDRAMA        > SDRAM0 PAGE 0 /* 32K word page */
   /*   SDRAMB        > SDRAM1 PAGE 0 /* 64K word page */
84 /*   SDRAMC        > SDRAM2 PAGE 0 /* 64K word page */
   /*   SDRAMD        > SDRAM3 PAGE 0 /* 64K word page */
86 /*   SDRAME        > SDRAM4 PAGE 0 /* 64K word page */
   /*   SDRAMF        > SDRAM5 PAGE 0 /* 64K word page */
88 /*   SDRAMG        > SDRAM6 PAGE 0 /* 64K word page */
   /*   SDRAMH        > SDRAM7 PAGE 0 /* 64K word page */
90 /*   SDRAMI        > SDRAM8 PAGE 0 /* 64K word page */
   /*   SDRAMJ        > SDRAM9 PAGE 0 /* 64K word page */
92 /*   SDRAMK        > SDRAM10 PAGE 0 /* 64K word page */
   /*   SDRAML        > SDRAM11 PAGE 0 /* 64K word page */
94 }

```