Experimental Demonstration of Adaptive Distributed Transmission Scheme for Indoor Wireless Networks

by

Cynthia M. Chow

Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering in
Electrical Engineering and Computer Science at the
Massachusetts Institute of Technology

August 22, 2002

Author _____
Department of Electrical Engineering and Computer Science
August 22, 2002

Certified by _____
Dr. Michael R. Andrews
Bell Labs Thesis Supervisor

Certified by _____
Prof. Gregory Wornell
M.I.T. Thesis Supervisor

Accepted by _____
Prof. Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

Experimental Demonstration of Adaptive Distributed Transmission Scheme for Indoor Wireless Networks

by

Cynthia M. Chow

Submitted to the
Department of Electrical Engineering and Computer Science

August 22, 2002

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Adaptive Distributed Transmission (ADT) is a novel scheme for transmission of data in a wireless network. ADT coordinates the transmissions of base stations in order to increase the overall throughput of the network. This gain is achieved through exploiting the space resource, by cooperatively compensating for the phase and amplitude effects of the scattering environment. Initial estimates expect throughput gains of almost four times that of traditional wireless networks. The dissertation describes in detail the first experimental demonstration of ADT in a small wireless network that can send data from base stations to mobile units in a changing environment, and establishes the feasibility of employing ADT to increase the capacity over larger networks.

Bell Labs Thesis Supervisor: Dr. Michael R. Andrews

M.I.T. Thesis Supervisor: Professor Gregory Wornell

Acknowledgements

I am deeply indebted to several people who have made significant contributions to my master's studies. I first would like to thank my parents, Alice and Bock Chow, for their love and encouragement. Their unwavering faith in me has always been a source of inspiration. Also, I am grateful to my sister, June, whose humor kept me sane.

I would also like to thank my Bell Labs mentor, Dr. Mike Andrews who provided the foundation of my work, and supported me throughout the entire process with valuable help and ideas. I feel very privileged for having had the opportunity to work with him. I am also grateful to my M.I.T. thesis supervisor, Professor Gregory Wornell who, from the start, has been extremely supportive of me. I would especially like to thank him for making the completion of my work possible through his encouragement and insight.

Additionally, I wish to thank Brendan Kao for his encouragement and time. He has always been there for me, and many insights arose from my discussions with him. Many times, he gave me the energy and optimism to persevere. Finally, I would like to thank Aref Chowdhury, who has been invaluable in making my progress as smooth as possible.

Contents

Chapter 1

1.1 Introduction

Wireless networks have traditionally utilized multiplexing in time, frequency or some combination of time and frequency (codes) to transmit data. These resources are limited for a given system, and close spatial proximity limits the re-usability of the same frequency or code due to co-channel interference [1-2]. In traditional frequency division multiple access (FDMA) consisting of hexagonal cells, acceptable performance is achieved if the interference power is at least 10 dB lower than the signal of the local transmitting base station. This performance is achieved by assigning each cell a different frequency from that of all adjacent cells. This frequency assignment scheme allows a maximum re-use factor of 1/7 with no interference [3].

In order to improve this re-use factor, numerous ideas have been proposed to intelligently manage interference. One way is to work around interference using a technique such as Code Division Multiple Access (CDMA). CDMA transmits different data on the same frequency, and manages interference by using pseudorandom codes. However, this often means there is a lot of interference, leading to a degraded signal-to-noise ratio (SNR) [2]. There have been several multi-antennae ideas proposed for beam-forming and nulling co-channel interference [4-8]. These systems can operate without any specific information of the channel and can provide increased capacity by creating greater path diversity [5]. Other systems are adaptive, and utilize pilot signals to monitor the channel [4,7,9]. All of the aforementioned ideas, however, associate all the antennae

with one base station. The Adaptive Distributed Transmission (ADT) scheme [3] proposes a way to extend the multi-antennae concept to a network of transmitters. Working in a flat-fading environment, the channel from each transmitting base station to each mobile position, can be characterized by a certain amplitude and phase. Thus, the entire network can be expressed as a matrix of complex numbers, with each element characterizing a link between a base station and a mobile. One method of using this information would be to tailor the interference received by each mobile [10]. The Adaptive Distributed Transmission (ADT) scheme [3] proposes using the matrix information to cooperatively transmit linear combinations of the messages destined for each mobile. By doing this over the entire network, the co-channel interference in the network will in fact cause the messages for each mobile to constructively interfere at that mobile's position, but destructively interfere at all other mobile locations. By periodically monitoring the channels through feedback from the mobiles, the bases can adapt to changes occurring in the network.

There are several fundamental conditions that limit the types of environments suitable for the application of ADT. The first is the time coherence of the channels. The scattering environment is constantly changing, and as a result, the communication channels are also constantly changing. If the rate of change is very fast, then the time coherence τ will be very small, and ADT will not be able to adapt quickly enough. Additionally, the channels over which ADT operates must be flat-fading. A flat-fading channel is one in which the signal bandwidth is much less than the channel's coherence bandwidth [11]; at any given point in time, the flat-fading channel can be entirely

characterized by a complex number. Note that signal bandwidths larger than this flat-fading bandwidth, $\Delta\omega$, can be handled by simply splitting them into flat-fading sub-channels. Initial estimates based on recent measurements, as well as standard wireless channel models predict that $\tau\Delta\omega > 1000$ is a conservative estimate for conditions in which the ADT scheme should be applicable [3].

Theoretically, 100 % efficiency in the re-use factor can be achieved in ADT systems. However, there are practical issues that prevent this theoretical limit from being attained. There are errors in measuring the channel due to delays and errors in channel measurement and estimation. Also, in a large network, it would be impractical to require the entire network to cooperatively transmit to all mobiles, including those very far away. In fact, one of the key ideas of ADT is that in a large or even infinite network, only local cooperation is required. Given these errors, a re-use factor of 2/3 is expected, while maintaining a 10 dB signal to noise ratio, which should be adequate for transmission of data. This is a factor of approximately four times more efficient than the aforementioned filling fraction of 1/7. Thus, it is expected that ADT can almost quadruple the current capacity for wireless networks.

The objective of this research was to demonstrate the basis for the ADT method in a real world wireless environment. Although the gains in network capacity from using phase control are well understood, its effectiveness in a real-world environment that constantly changes is less so. One goal of this research was to explore if it was indeed possible to track changes of phase and amplitude, and compensate for them in a slowly varying environment. Furthermore, for ADT to have commercial application, an ADT

network must be able to be built out of readily available, inexpensive hardware. Thus, another equally important goal was to study the performance of a real ADT system implemented using off-the-shelf components.

The outline of the thesis is as follows: Chapter 2 presents the background and theory of ADT. Chapter 3 then proceeds to outline the design requirements and hardware. Chapter 4 provides a detailed description of the prototype system, as well as the challenges faced in implementation. Chapter 5 goes on to analyze the performance results in a characteristic indoor environment. Finally, Chapter 6 concludes with a summary of the research performed, as well as a discussion on areas for future work.

1.2 References

1. T. S. Rappaport, *Wireless Communications: Principles and Practice* (Prentice-Hall, Englewood-Cliffs, NJ, 1996).

2. A. J. Viterbi, *CDMA: Principles of Spread Spectrum Communication* (Addison-Wesley, Reading, MA, 1995).

3. B. Shraiman, M. Andrews, and A. Sengupta, Technical Report No. ITD-00-40485E, Bell Labs, Lucent Technologies (unpublished).

4. J. H. Winters, "On the Capacity of Radio Communications Systems with Diversity in a Rayleigh Fading Environment," *IEEE Journal on Selected Areas in Communications,* **SAC-5**, 871 (1987).

5. J. H. Winters, J. Salz and R. D. Gitlin, "The impact of antenna diversity on the capacity of wireless communication systems," *IEEE Transactions on Communications*, **42**, 1740 (1994).

6. J. S. Thompson, P. M. Grant and B. Mulgrew, "Smart antenna arrays for CDMA systems," *IEEE Personal Communications*, **3**, 16 (1996).

7. H. R. Karimi, M. Sandell and J. Salz in *Comparison between transmitter and receiver array processing to achieve interference nulling and diversity* (IEEE, Osaka, Japan 1999).

8. R. M. Buehrer, A. G. Kogiantis, S. Liu, J. Tsai, and D. Uptegrove, "Intelligent Antennas for Wireless Communications Uplink," *Bell Labs Technical Journal*, **4**, 73 (1999).

9. G. J. Foschini and M. J. Gans, "On Limits of Wireless Communications in a Fading Environment when Using Multiple Antennas," *Wireless Personal Communications*, **6**, 311 (1998).

10. S. Shamai, "On Information Theoretic Aspects of Multi-Cell Wireless Systems," *Bell Laboratories Research Seminar* (April 25, 2002).

11. P. Hande, L. Tong and A. Swami, "Flat fading approximation error," *IEEE Communications Letter*, **4,** 310 (2000).

Chapter 2

Adaptive Distributed Transmission: Theory

2.1 ADT

Consider a wireless network consisting of M mobiles, at positions x_i and N base stations, in a triangular lattice, with vertices r_a, transmitting in all directions as shown in Figure 2.1. All communications from the base stations to the mobiles occurs over one frequency (or code). Assuming that the network is operating within the flat-fading bandwidth criteria, there is a transmission kernel between the a-th base station and the i-th mobile:

$$K_i^a(t) = \left|x_i - r_a\right|^{-2} e^{ik(x_i - r_a) + i\varphi_{ia}(t) - \eta_{ia}(t)} \tag{1}$$

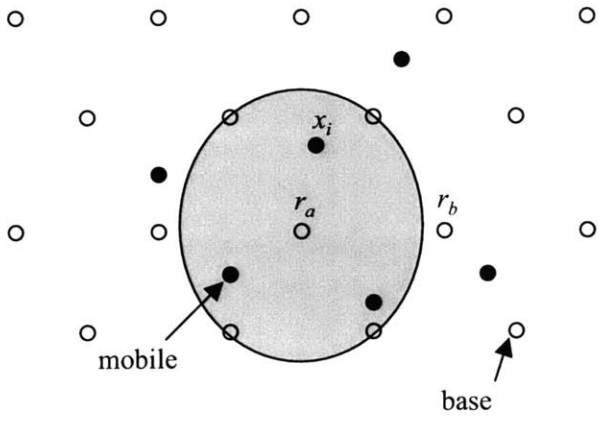


Figure 2.1. Base station at position r_a and mobile x_i

This transmission kernel captures the effect of multiple scattering with a random phase φ_{ia} and a Rayleigh fading exponent η_{ia}. Thus, the matrix K characterizes the effects of the network on the phase and amplitude of transmissions. Knowing K allows the base stations to pre-compensate their transmissions in order to cancel the effects of the network. Note that one can expect

$$\left\langle K_i^a(t)K_b^j(0)^*\right\rangle = \delta_{ij}\delta_{ab}\frac{Ae^{-t/\tau}}{\left|x_i - r_a\right|^4} \tag{2}$$

where τ determines the correlation time. In general, τ depends on the distance between a base and a mobile [1]. It is important that τ is sufficiently long for the matrix K to be measured.

Given the K matrix, pre-compensation involves all the base stations sending different linear combinations of messages to the mobiles. If messages $m_i(t)$ are to be transmitted to the mobiles, then every base station sends

$$s_a = L_a^i m_i \tag{3}$$

Each element L_a^i determines the phase and amplitude of each message m_i. These modified messages are then linearly combined, forming the transmission that base station a sends. The mixing matrix L should be chosen to minimize the crosstalk in the network. The crosstalk for each channel is determined by

$$C = \max_i \left\{ \frac{\sum_{j \neq i} \left| \sum_a K_i^a(t) L_a^j m_j \right|^2}{\left| \sum_a K_i^a(t) L_a^j m_i \right|^2} \right\} \tag{4}$$

It is clear that if $L(t) = K^{-1}(t)$ (or in the case where M < N, $L(t)$ is the pseudo-inverse of $K(t)$), then the off-diagonal terms would go to 0, yielding $C = 0$ [1].

2.2 Estimated Performance for ADT

2.2.1 Sensitivity to Error

In a real system, errors are unavoidable. Due to noise in the network, as well as the changing channel, there will be measurement error in the channel estimation. Furthermore, in the extended network, truncation also introduces error. It is important that the effects of errors in the ADT method be bounded by a reasonable limit, if it is to be used in a real environment. Assume that at best, we can only set $L = T(\hat{K}^{-1})$, where $\hat{K}$ is an estimation of the network kernel K. The operation $T(\ )$ denotes the truncation of the matrix, where elements below the bound γ are set to zero. Then the expected ratio of signal to noise is [1]

$$\langle C_i \rangle = q_i^{-2} \sum_{j \neq i} q_j^2 \left\langle \left| \sum_a \delta K_i^a(t) L_a^j + K_i^a(t) \delta L_a^j \right|^2 \right\rangle \tag{5}$$

Expanding the error into its contributing components, it can be seen that there are two main contributing factors: $\delta_{\Delta} K_i^a(\Delta t)$, which is the prediction error, and error due to truncation. Initial work has indicated that the effect of estimation and truncation errors is bounded, while still allowing reasonable system performance [1].

Even with bounded error, it is important that the network kernel matrix be stable, with small eigenvalues, since large eigenvalues would amplify estimation errors. Simulations have shown that proper spatial distribution of mobiles should prevent instability and large eigenvalues in the network [1]. The constraints on spatial distribution are discussed in the following section.

2.2.2 Spatial Constraints For Stability

As mentioned above, it is key that the network kernel be non-singular, with small eigenvalues, so as not to amplify errors in matrix estimation. Simulations have shown that these conditions are satisfied if clustering of mobiles in space is avoided [1]. One method investigated was to constrain the number of mobiles per hexagonal cell to one. This does indeed prevent clusters of mobiles from forming, given that there is special handling of the boundary conditions. However, in an unconstrained network, there is a finite probability of clusters forming. Thus, to satisfy this constraint in the general case, clusters of mobiles could be re-allocated to different channels such that they no longer form spatial clusters.

2.2.3 Effects of Truncation

ADT is in theory applicable to very large, and even infinite networks (where the number of base stations and mobiles is very large.) However, when the network grows beyond a certain size, the mixing matrix L may contain very large elements. This is because a large network matrix K may be sparse, and have elements with very small amplitudes, if the distance between that mobile and base is quite far. However, the inverse of K may not be sparse and can contain matrix elements with very large amplitudes. This is likely to occur if mobiles far away from a base station are considered in the cooperation. This results in a base station using a lot of power to transmit messages to mobiles that are very far away. This is undesirable in a real system. However, in ADT, local cooperation allows elements in the matrix K whose amplitude is less than a truncation limit γ to be set to zero. The effect is that for each base station, only mobiles in a local area need to be considered. However, since truncation introduces error, it will create some residual crosstalk. Initial findings concluded that truncation error dominates when measurement error is less than -10 dB. Furthermore, it is found that about 10 dB suppression of crosstalk can be achieved with $\gamma = 0.1$ (where γ is the truncation factor), for frequency re-use factors of about 2/3 [1]. In simulation, this truncation factor was equivalent to each mobile monitoring the bases in the nearest and next nearest coordination shells.

2.2.4 Sensitivity to Measurement Error

Intrinsic error in the measurement of the network kernel elements is caused by several factors. One contribution is the finite signal to noise ratio in the system. Signal levels weaker than the background noise will not be measured accurately. A more fundamental limit comes from the delay in measuring the channel. There is an unavoidable delay due to the time for the mobiles to measure the channel and report back to the base stations. From Equation (5), the component $\delta_\Delta K_i^a(\Delta t)$ is the intrinsic report delay. It has a variance $\left\langle \delta_\Delta K_i^a(\Delta t)\delta_\Delta K_j^b(\Delta t)^* \right\rangle = \delta_{ij}\delta^{ab}\sigma(\Delta t)\left\langle \left|K_i^a(\Delta t)\right|^2 \right\rangle$ where

$$\sigma(\Delta t) \equiv \left\langle \left|\delta \ln K_i^a(\Delta t)\right|^2 \right\rangle \tag{6}$$

is the variance in forecasting the phase and amplitude of the transmission kernel. The contribution to crosstalk is bounded by $\beta_i\sigma(\Delta t)$ where

$$\beta_i = q_i^{-2}\sum_a \left|K_i^a\right|^2 (L + q^2 L)_{aa} \tag{7}$$

is the error sensitivity factor. In the simulations performed, β_i was always $o(1)$, implying modest error amplification [1].

2.2.5 Expected Environmental Conditions

As mentioned before, ADT is expected to work in indoor environments, or more generally, in environments which have $\tau\Delta\omega > 1000$. This number can intuitively be thought of as a measure of the suitability of the environment. τ describes the time coherence of the channels, and $\Delta\omega$ is the total flat fading bandwidth available. Some of this bandwidth will be sacrificed to pilots. It is easy to see that if the time coherence is very large, meaning that the channels hardly change over time, then channel measurements can be less frequent, requiring less bandwidth overhead. Conversely, if τ is small, then the channels are changing quickly, and a large amount of bandwidth will need to be sacrificed to pilots. Thus, if the product of the two is large enough, meaning that there is more than enough flat-fading bandwidth to send the necessary pilots, then ADT can adapt.

There are two types of scattering environments that are expected to satisfy this constraint. For simplicity, only the phase randomness model for channels in these environments was considered. However, the same argument applies for the Rayleigh fading component. The two types of regimes are the phase drift regime and the phase order regime. In the phase drift regime, large changes in phase occur in a coherent fashion. For the phase order regime, the phase is constant for long periods of time, and fluctuates close to an average value whose variance behaves like the phase drift regime [2-4]. Using this information, as well as well-known channel models, estimates for flat-fading bandwidth and time coherence were derived. Reasonable estimates for $\Delta\omega$ were 100 kHz, with τ around 10 ms, in these scattering environments. These conditions

would allow adequate performance (10 dB SNR) with less than 10 % pilot overhead [1]. Initial calculations based on recent scattering environment experiments suggest that an indoor environment, such as that of Bell Labs, is likely to fall within this category [2-4].

2.3 References

1. B. Shraiman, M. R. Andrews, and A. Sengupta, Technical Report No. ITD-00-40485E, Bell Labs, Lucent Technologies (unpublished).

2. M. Stoytchev and H. Safar, Technical Report No. ITD-99-38447P, Bell Labs, Lucent Technologies (unpublished).

3. M. Stoytchev and H. Safar, Technical Report No. ITD-99-38438E, Bell Labs, Lucent Technologies (unpublished).

4. M. R. Andrews, P. P. Mitra and R. deCarvalho, "Tripling the capacity of wireless communications using electromagnetic polarization" *Nature*, **409**, 316 (2001).

Chapter 3

Prototype 2x2 ADT Network

3.1 Overview

The goal of this dissertation was to demonstrate a prototype ADT system in a real wireless network. There were several challenges in creating a system that operates in a real environment. One of these challenges was operating in real-time. Another was maintaining acceptable performance with the limitations of off-the-shelf components. As a result, a major part of this research focused on the design and implementation aspects of such a system.

The prototype network consisted of four nodes as shown in Figure 3.1. Two nodes were base stations and two were mobiles. The base stations transmitted data and

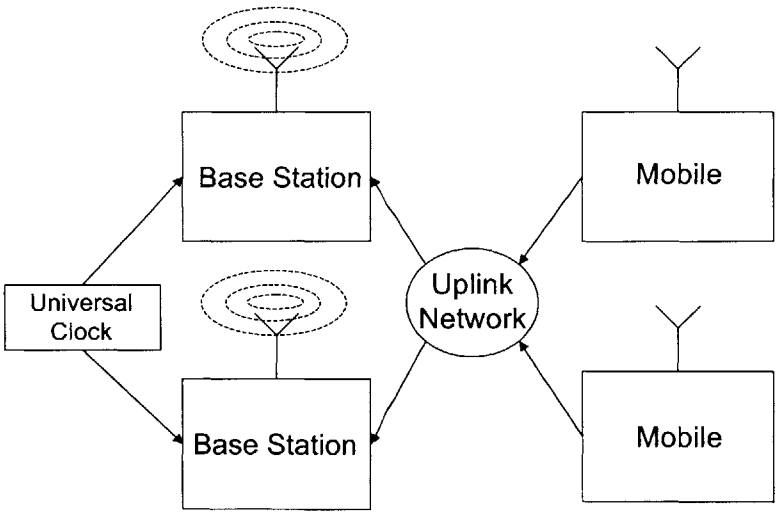


Figure 3.1 Prototype ADT Network.

channel estimation pilots to the mobiles over the wireless channel. The mobiles in turn received the data and pilots, and periodically reported the channel measurements to the base stations. The base stations then used these measurements to appropriately adapt their next transmissions. In this prototype, the uplink was a wired network, simulating an out-of-band link from the mobiles to the base stations. However, in a commercial implementation, the uplink network would also be wireless.

The base stations and mobiles were all implemented on Texas Instruments DSP evaluation boards, and personal computers (PCs). There were also two Radio Frequency (RF) transmitters and two RF receivers, for the wireless downlink. A Local Area Network (LAN) was used as the out-of-band uplink. A signal generator served as the Universal Clock.

This chapter describes the general operation of the prototype network, including the major functions that the base stations and mobiles perform, as well as any requirements they impose. The chapter goes on to describe key elements of the hardware resources, as well as the manner in which these resources were utilized.

3.2 Base Station

The two base stations were responsible for transmitting data messages to the mobile units. In order to do so successfully, they coordinated their transmissions. In the network with two base stations and two mobiles, there are two messages, one intended for each mobile. Each base station simultaneously sent a linear combination of all messages. Since the mixing matrix describes amplitude, as well as relative phase differences between base transmissions, assuming that they began at the same time, it

was imperative that the base stations transmissions be simultaneous. This critical synchronization of the base stations was accomplished by referencing a Universal Clock. Specifically, synchronization can be characterized by three attributes: phase (φ), frequency (ω_i), and time (t). The criteria for each are shown in Equations (1)-(3).

$$\frac{d\varphi}{dt} << \frac{1}{\tau} \tag{1}$$

$$\Delta\omega t_U << 1 \tag{2}$$

$$\Delta t / t_s << 1 \tag{3}$$

For Equation (1), $\frac{d\varphi}{dt}$ is the rate of phase drift between the RF local oscillators, and τ is the time coherence of the channel. This constraint means that the phase drift over the channel should be much faster than any phase drift seen due to the RF local oscillators. Generally, this can be accomplished through actively compensating for a known drift. In the specialized case of the prototype system, this factor was removed by using the same local oscillator for both base stations. Equations (2) and (3) deal with the variations in codec sampling clocks. In Equation (2), $\Delta\omega$ is the difference in sampling frequency between any pair of base station codecs, and t_U is the period of the Universal Clock. Their product must be small enough that over the duration of a message, all base stations will finish the transmission before the first sample of the next message begins. Equation (3) relates a similar constraint: Δt is the difference between any two base stations in the time it takes to transmit a message. Their difference should be much less than t_s, the

time it takes to transmit one sample at the ideal sampling rate. The codecs used satisfied these constraints.

Embedded in the messages sent to the mobiles, the base stations also sent pilots. One pilot was used for the mobiles to recover timing with the base station. The other pilots were used for estimating the network matrix. Periodically, the base stations received reports about the channels among all the mobiles and all the base stations. These reports formed the network matrix. All the base stations must know the current network matrix so that they can calculate the inverse of the network matrix. Once the inverse was known, a base station used the row concerning the channels between it and the mobiles to mix its next outgoing transmissions. An example of how base station 1 transmitted its messages is shown below in Figure 3.2.

3.2.1 Messages

The messages that the base stations transmitted were arrays of n symbols. These symbols were chosen at random from the data constellation. Interspersed with the data

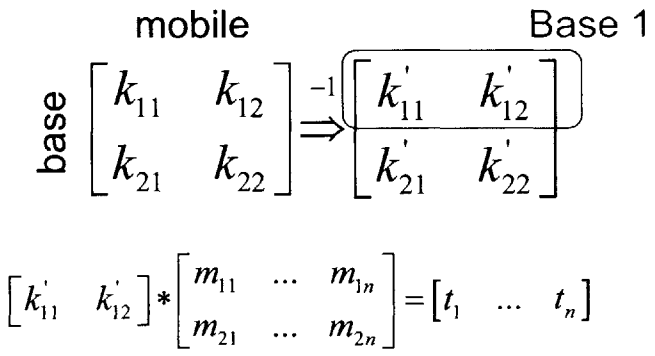


Figure 3.2 Each base station uses the row in the inverse of the network matrix to mix the messages to send to the mobiles.

were the three pilot signals described above.

In order to transmit each message, the base stations converted the messages into analog waveforms. This was accomplished by modulating them onto pulse shaping waveforms. The pulse shaping waveforms were designed to be time-limited, so that they occupied a finite number of samples. They were also band-limited, which was essential for several reasons: the hardware had a maximum sampling rate, and the channels had a finite flat-fading bandwidth. The pulse-shaping waveforms needed to satisfy both constraints. In the system, the hardware was the limiting factor on bandwidth due to the limited sampling rate as well as clock synchronization issues; these aspects will be discussed in more detail in section 4.3.2.1 Clock Drift. The bandwidth of the pulse shaping waveforms in turn determined the realized bandwidth of the downlink.

3.2.2 Pilots

There were two types of pilots used in the system. The first was the channel estimation pilots. These pilots essentially sent the identity matrix over the network, i.e., the estimation pilots transmitted from the base stations for the prototype network formed the two-by-two identity matrix. Physically, transmitting this matrix occurs in the time dimension: both base stations had two messages to send simultaneously, which formed a 2xn matrix, where n was the number of data symbols in each message, to be transmitted over a time frame. At the front of this matrix lay the 2x2 identity matrix, which constituted the pilot matrix. Upon transmission, each column of the identity matrix was sent at the appropriate time slot. The pilot matrix was not mixed, like the rest of the data,

and simultaneous transmission of the messages ensured that the pilots were sent simultaneously as well.

As a design note, the pilot matrix did not have to be the identity matrix. Any rank 2 matrix of known values could have been used. The best matrices, however, have condition 1, and are unitary, so that no particular channel is favored. Even fancier, the pilots could have been complex, so that the in-phase (I) and quadrature (Q) channels on the RF hardware would put out even power. However, the identity matrix satisfied the requirements, and was convenient, so it was used. As another design note, the columns of the pilot matrix did not have to be placed in the front of the message, nor did they have to be directly adjacent to one another. Putting the pilots at the front of the message was arbitrary, and the columns were separated by blanking symbols to improve performance. The performance aspect will be discussed in section 4.3.1 Pilots.

The second type of pilot used was a timing recovery pilot. The timing pilot carried more power than the rest of the data symbols or matrix pilots to aid the mobiles in recovering sampling timing with the base station transmissions. The timing recovery pilot was included in each message, and was linearly combined between messages, just like data.

3.3 Mobiles

The mobiles received data over the wireless channels from the base stations. Receiving this signal first involved demodulating the RF signal down from the carrier. This is generally accomplished using a local oscillator that is the same frequency as the transmit carrier. However, there will be small frequency differences between transmit

and receive local oscillators that would appear as carrier phase drift. Thus, the constraint on this follows the phase constraint in Equation (1). In the general case, the phase drift would be removed through signal processing methods. However, to simplify the prototype, and to ensure that observed phase drift was due to the channels, the transmit and receive local oscillators were all the same clock. In order to receive the message, the mobiles only needed to recover timing with the base stations, since the coordinated transmissions from the base stations constructively interfered over the network to arrive at each mobile with its intended message. To recover timing, a mobile had to discover where the first sample of the transmission occurred in the received waveform. This timing recovery had to be exact to the nearest sample, or else the received data would not decode properly into constellation symbols.

After the mobile decoded the received data, it periodically reported the channel measurements, which were demodulated along with the data. Since the channel estimation pilots were sent without mixing over the channel, the received values directly reflected the effect that the wireless channel had over any transmission within the same flat-fading band as the pilot waveforms. These channel measurements were then sent over the uplink to all the base stations.

3.4 Hardware Components

The prototype network was built entirely out of readily available hardware. This included Texas Instruments 6701 DSP evaluation boards, Pentium-based PCs, RF transmitters and receivers (with quadrature mixers), and a standard signal generator.

Greater details about the hardware architecture and capabilities can be found in the TI Reference Manuals and User Guides [1-5].

3.4.1 TI 6701 DSP Evaluation Board

The major parts of the evaluation board used for this project were the 6701 Texas Instruments DSP processor, external memory, a serial port, the Direct Memory Access (DMA) module, an audio codec, an analog line in and line out interface, and a Host Port Interface to the PCI bus as shown in Figure 3.3.

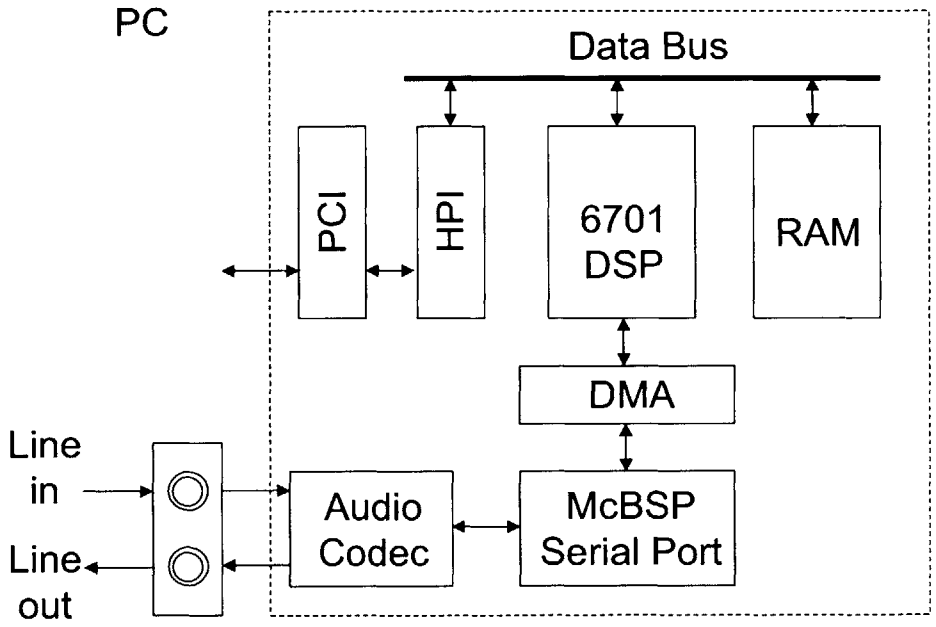


Figure 3.3 Block diagram of the major components on the TI C6701 DSP evaluation board.

3.4.2 C6701 DSP Processor

The C6701 is a fixed and floating point DSP processor that runs at 133 MHz. It runs under the Texas Instrument's DSP/BIOS, which is a real-time operating system. This operating system provides several useful features for programming applications that must meet real-time deadlines. One of these is the hardware interrupt. Hardware interrupts (IRQs) allow peripheral components to notify the DSP of some time-sensitive task that must be performed. Usually, this task is associated with an Interrupt Service Routine (ISR) that the DSP must run to service the peripheral. In operation, an IRQ is handled by an assembly routine with a special prologue to save the appropriate registers, and jump to the ISR. Upon program initialization, an ISR vector is created that tells the DSP where each ISR is located. Implementation with DSP/BIOS wraps this task into a configurable hardware interrupt manager (HWI manager). The HWI manager handles low-level calls to ISRs, and allows ISRs to be written in C, rather than assembly.

Another real-time tool provided with DSP/BIOS are the prioritized software threads that allow easy multi-threaded programming. DSP/BIOS contains a Software Interrupt Manager (SWI manager) that provides a graphical interface for thread configuration, and a C callable Application Programming Interface (API) for invoking threads. The SWI model is based on the hardware interrupt concept, where software interrupts can be posted. Posting a thread causes that thread to run when it is the highest priority task waiting for execution. Thread priorities allow the most time-sensitive task to preempt those which do not have such stringent real-time deadlines. SWIs have up to 13 levels of priorities, all of which are lower than that of HWIs.

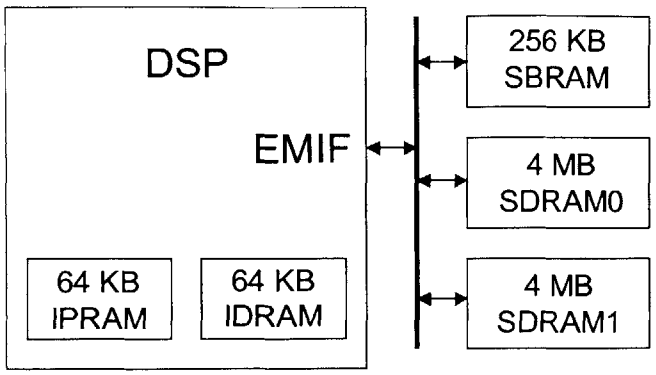
The cycles needed to execute code present another limitation in meeting real-time deadlines. TI provides C callable routines for performing popular DSP tasks, such as FFTs and vector multiplications. These routines are optimized for the C6701 architecture to execute in the minimal number of cycles. The C compiler for the C6701 DSP is another powerful optimization tool. It has the ability to software pipeline code, as well as provide compiler feedback for optimization. This allows for an otherwise unattainable high rate of computation.

3.4.3 Fixed vs. Floating Point

The 6701 has both fixed- and floating-point capabilities. However, the prototype system was limited to be intrinsically a fixed-point system due to the fact that the codec operates in fixed-point mode. Furthermore, most of the specialized signal processing algorithms operate on fixed-point data. Thus, the system is designed for signed 16-bit integers. This means that extra care must be taken at all steps to avoid overflow of mathematical computations. Furthermore, quantization error will be present throughout the system. In this regard, when extra precision was needed for internal algorithms, the floating-point hardware of the 6701 was used to speed computation without these limitations.

3.4.4 Memory

The DSP itself has 128 KB of on-chip memory. However, this is not enough to load and run the base station and mobile programs. Thus, the evaluation board also provides an additional 8 MB of fast RAM and 256 KB of slow RAM, accessible through



Section	Address
IPRAM	0x00000000
IDRAM	0x80000000
SBRAM	0x00400000
SDRAM0	0x02000000
SDRAM1	0x03000000

Figure 3.4 Internal and external RAM, listed with the address space of MAP1.

the External Memory Interface (EMIF) on the DSP. This memory has two possible configurations – MAP0 and MAP1. For the prototype demonstration, MAP1, illustrated in Figure 3.4, was used. The different sections of memory have different uses. The on-chip memory (IDRAM and IPRAM) is the fastest, so it is used to hold run-time variables as well as executing code. The memory sections in the external RAM (SDRAM and SBRAM) are slower, but can be loaded with large data sets. This was important since much computation was saved in the various base station and mobile algorithms by storing pre-computed look-up tables and waveforms in this section of memory.

3.4.5 McBSP Serial Port

The evaluation board provides several interfaces for exchanging data between peripheral devices and the DSP. One of these interfaces is the McBSP serial port. This is used to exchange data between the DSP's memory and the audio codec. The serial port is fully programmable, and can generate an interrupt on receipt or transmission of a sample.

The serial port interfaces with the DSP through several registers. There are two configuration registers, the Primary Control Register (PRICTL) and the Secondary Control Register (SECCTL). There is a third status register. The interface for data exchange also consists of two registers which are mapped to DSP memory: the Data Receive Register (DRR) and the Data Transmit Register (DXR). These correspond to addresses 0x018C0000 and 0x018C0004 in the DSP's memory. The DSP can thus receive a sample by reading the DRR on an interrupt, and send a sample by writing to the DXR after the prior transmission is completed.

The serial port reads data from the codec at the programmed sampling rate. It is the responsibility of either the DMA channel servicing the serial port, or the DSP to read the incoming data before the next sample arrives. If the data is not removed from the register before the next data arrives, an error event is triggered. This error event can be checked by looking at the condition bits in the status register. Code in Appendix B and Appendix C describe the configuration of the PRICTL and SECCTL registers for the base station and mobile.

3.4.6 Direct Memory Access

The Direct Memory Access (DMA) module manages DMA channels. A DMA channel can transfer data between any addresses in memory. This means that for memory mapped peripherals, such as the serial port, a DMA channel can service it. A DMA channel is configured to read data from a source address and write it to a destination address. Each channel has the ability to auto-increment the source and destination addresses after each transfer. This means that a DMA channel can automatically copy

contiguous chunks of memory from one location to another, or it can repeatedly read data from a register and place it into an array in memory. A DMA transfer is completed once all elements have been transferred. The number of elements to transfer is set by the count value in a register. Both the count value as well as source and destination addresses can be set at initialization, as well as by global registers. If global registers are used, then the DMA channel can reinitialize subsequent transfers without DSP intervention. This is very important since auto-initialization occurs faster and it allows the DSP to continue without interruption.

An additional feature of the DMA module is that each channel can synchronize reads or writes to events. There are several pre-programmed synchronization events, including a receive and transmit interrupt from the serial port. The DMA can also generate its own synchronization events, such as interrupts to the DSP.

3.4.7 Crystal Semiconductor Audio Codec

The DSP board contains several peripherals, including an audio codec. The audio codec communicates with the DSP through the serial port. This codec is very flexible, with a programmable sampling rate, data format, input gain and output attenuation.

Signal input and output of the codec occurs over the Line In and Line Out interfaces. Before the signal reaches the codec, it is passed through low pass filters that remove frequency components above half the fastest sampling rate. The sampling rate, provided by two crystal oscillators, can be set to discrete values ranging from 8 kHz to 48 kHz. The data format is similarly programmable. The codec provides different methods for quantizing the signal. The linear companding format, in stereo mode was used in the

prototype. In this configuration, the codec converts each channel of a stereo analog signal into two signed 16 bit integers, ranging from –32,767 to 32,767. The 16 bits for the right channel occupy the top 16 bits of a 32-bit unsigned integer, and the left channel occupies the bottom 16 bits. The availability of stereo is convenient for the prototype network, since the data consists of real and imaginary components. The right channel is used to carry the imaginary component of the signal, and the left channel is used to carry the real component of the signal.

The codec is also capable of applying a gain to the input signal and attenuation on the output signal. The gain and attenuation is programmable in 1.5 dB increments. This is useful since the RF Receive Module provides an attenuated signal, and the Line In interface to the codec applies a 6 dB attenuation. These attenuation factors can be compensated for by boosting the input gain of the codec. At 0 dB gain, the average noise was rated to be about 4 bits, with a maximum of about 6 bits.

3.4.8 Using the DMA to Service the McBSP Serial Port and Codec

One of the key building blocks in the implementation of the prototype system was using the DMA to service the codec. This application involves most of the hardware features described above, and was a major component used throughout the entire system. The application uses two DMA channels to service the input and output of the audio codec. One DMA channel receives samples from the codec and one writes samples to the codec, both operating through the serial port as shown in Figure 3.5.

On the receive channel, when the codec generates a sample, the serial port transmits that to its DRR and generates a Data Receive Event, which the input DMA

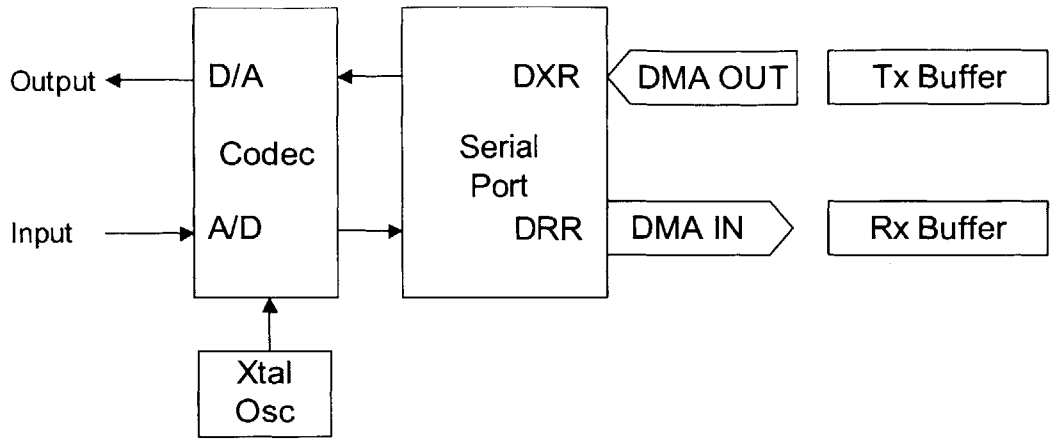


Figure 3.5 Using the DMA to service the Audio Codec through the Serial Port.

channel sees. The DMA channel then transfers this data to an array in DSP memory, and increments the array pointer to the next available address. When a certain number of samples have been received, the DMA channel generates an interrupt to the DSP, which then services the interrupt, clearing the interrupt condition. Note that the condition must be cleared, and the DMA restarted before the first sample of the next block can be received. The DMA must be configured to auto-initialize itself in order to meet this timing constraint. The clearing of the interrupt condition allows the DMA to re-initialize itself before the arrival of the next sample. After the DSP clears the interrupt condition, it is free to process data until the next DMA interrupt. The timing for this process of events is shown in Figure 3.6.

Sending an array of samples is a very similar operation, except the direction of data flow is in the other direction. On completion of a DMA transfer to the DXR, the DMA interrupts the CPU, which passes it an array of new data to transmit. The DMA writes a sample from this array to the DXR every time the serial port has signaled that it

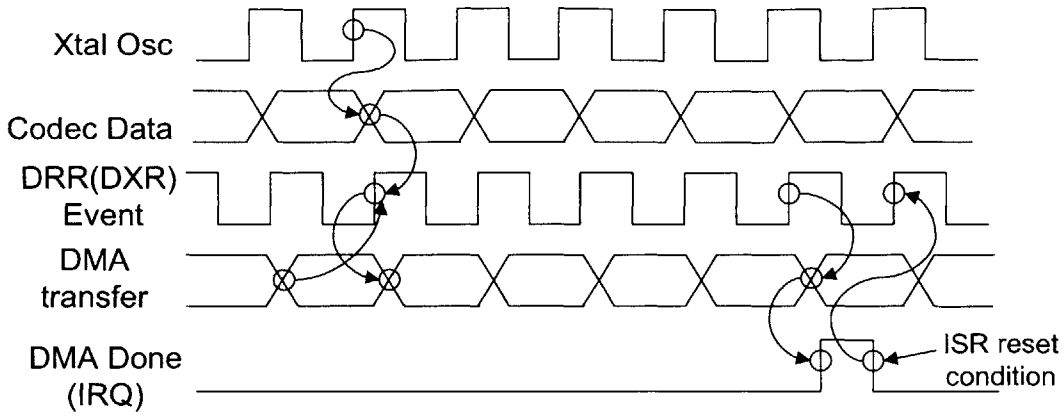


Figure 3.6 Timing diagram of DMA servicing Audio Codec.

wrote the previous sample to the audio codec. In this way, the DMA can be used to service the serial port, which allows larger periods of uninterrupted time for the DSP, resulting in faster performance. The source code for configuring the DMA, serial port, and audio codec for input and output are available as reference in Appendix B and Appendix C.

3.4.9 PCI Data Bus

The PCI data bus connects the DSP board to the host PC. The PCI data bus was used extensively to communicate between the board and host through the Host Port Interface (HPI). The HPI allows the host PC to read and write the DSP's on-board memory. On the DSP side, this is completely transparent. On the host PC side, TI provides Windows drivers which take care of the low level details of communication through the PCI bus, and wraps this functionality into C-callable functions. When the PC wants to access DSP memory, the host application calls the appropriate function. This function call accesses the Windows driver, which communicates with the HPI interface

on the DSP board. The HPI then accesses the appropriate portion of DSP memory, and performs the requested function.

3.4.10 Host PC

The host PCs for the DSP boards are Celeron-based personal computers. Each computer has a Celeron processor that ranges from 600 MHz to 800MHz. Three of the computers have 128 MB of RAM, and one of them has 318 MB of RAM. Each of these PCs have an Ethernet card, capable of transmitting data at up to 100 Mb/s. All the PCs are running Windows NT 4.0.

3.4.11 Transmit/Receive RF Modules

The Transmit and Receive Modules are responsible for modulating the baseband analog waveforms up to an RF frequency carrier, and transforming them to radio waves. The Transmit Module (TM) consists of an RF transmitter, a quadrature mixer, and an amplifier. The first stage of the TM is the quadrature mixer. When the base station unit sends a stereo analog waveform to the TM, this signal is split into the two channels and sent to the mixer. The mixer modulates the real channel onto the I carrier and the imaginary channel onto the Q carrier. The carrier is provided by a local oscillator that outputs a cosine wave with tunable frequency. The local oscillator's waveform is split into two equal powered duplicates, and one is phase shifted by 90°. After mixing, the signals are combined and amplified, then sent out over the transmit antenna. The transmit antenna was an omni-directional dual-band antenna designed for wall mounting. It was capable of transmitting at 868 MHz to 890 MHz, and 1.7 GHz to 1.9 GHz. The

carrier occupied the lower band, and was usually set to about 880 MHz. The antenna was rated for up to 50 W of power, however, the mixers and amplifiers used had a maximum rating of 10 dBm, so the baseline transmit power used in the system was approximately 2 dBm (1.58 mW). A schematic of the TM hardware is shown in Figure 3.7.

The Receive Module (RM) consists of hardware that is complementary to the Transmit Module: a receive antenna, an amplifier, and a mixer. Some additional low pass filters were also added. The receive antenna is a dual-band omni-directional wall-mount antenna, designed to received frequencies in the system band of 868 MHz to 890 MHz, as well as in the 1.7 GHz to 1.9 GHz band. The signal received from the antenna is first attenuated, then passed to an amplifier. The amplifier then passes the signal to the

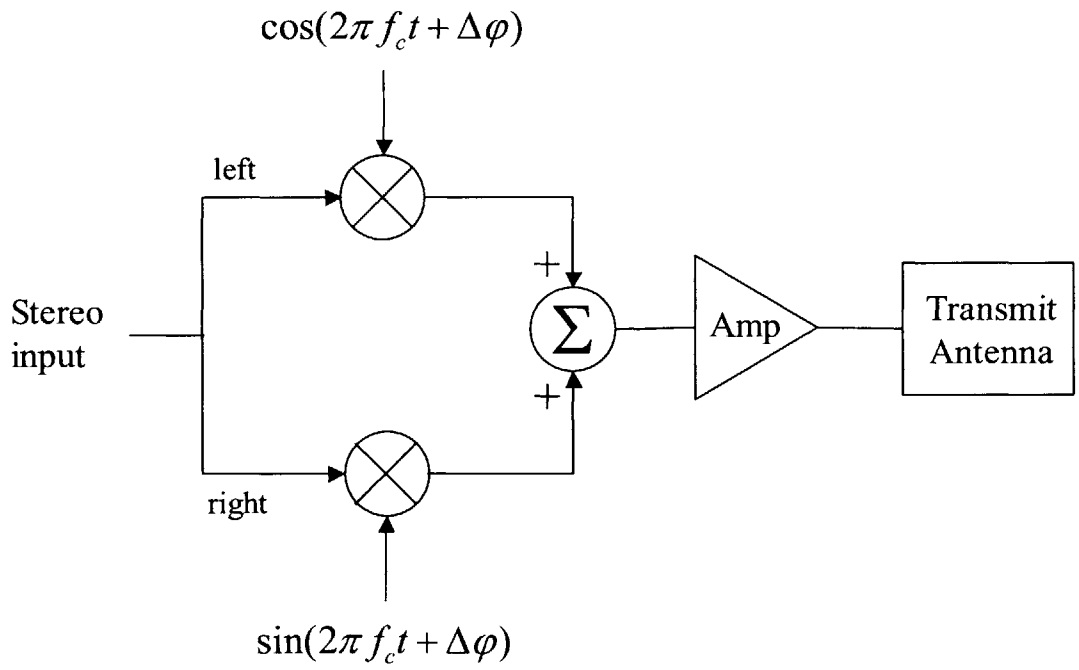


Figure 3.7 Block diagram of RF Transmitter.

mixer, which splits the signal into two. The mixer, as in the TM, has a local oscillator, which it uses to create an in-phase carrier and a quadrature carrier. The in-phase and quadrature waveforms are used to demodulate the two copies of the incoming signal. The result is first passed through 2 MHz low pass filters to remove extraneous frequencies, then recombined into a stereo signal. This stereo signal is then passed to the DSP board as shown in Figure 3.8. As mentioned in section 3.3 Mobiles, to satisfy the constraint that the phase drift between the transmit and receive clocks be very small, the same local oscillator was used for all transmit and receive hardware.

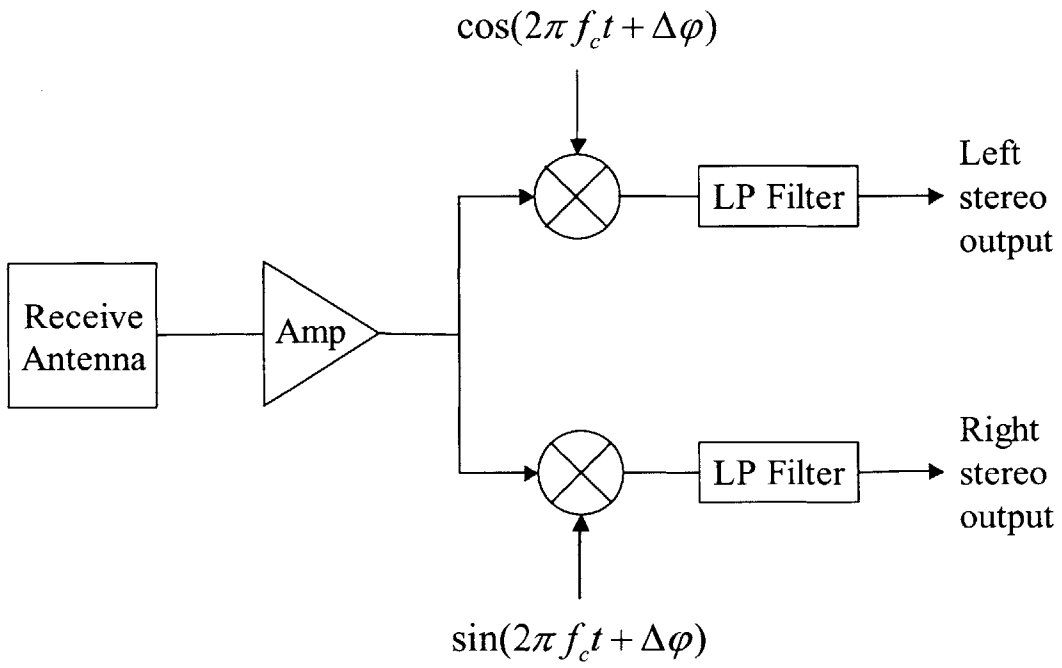


Figure 3.8 Block diagram of the RF Receiver.

3.4.12 Signal Generator

The signal generator is a standard waveform generator. It has three types of waveforms: a sine wave, a square wave, and a triangle wave. The square waveform is used as the global base station clock.

3.5 References

1. "TMS320C6000 Peripherals Reference Guide," *Texas Instruments TMS320C6000 Code Composer Studio Manuals* (2001).

2. "TMS320C6701 Evaluation Module Technical Reference," *Texas Instruments TMS320C6000 Code Composer Studio Manuals* (2001).

3. "TMS320C62x Peripheral Support Library Programmer's Reference," *Texas Instruments TMS320C6000 Code Composer Studio Manuals* (2001).

4. "TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide," *Texas Instruments TMS320C6000 Code Composer Studio Manuals* (2001).

5. "TMS320C6000 Chip Support Library API Reference Guide," *Texas Instruments TMS320C6000 Code Composer Studio Manuals* (2001).

Chapter 4

Adaptive Distributed Transmission: Design and Implementation

4.1 Wireless Message Protocol

This section discusses the design of the modulating waveforms, the data constellation and message format for the messages sent to the mobiles over the wireless downlink.

4.1.1 Pulse Shaping Waveforms

The design of the pulse shaping waveforms required considering several issues. Their bandwidth was limited by the physical constraints of the hardware. However, since this was a real system, the waveforms needed to be mostly time-limited so that truncation would not cause adverse effects. One set of waveforms that satisfied this property of time and band limitation was the discrete prolate spheroidal functions, also known as *slepians* [1]. The slepians also had additional characteristics that make them the waveforms of choice. One property was that a set of slepians can be chosen to satisfy a certain time and band restriction [2], such that they are all orthogonal to each other. In theory, this was ideal, since each orthogonal waveform can carry a symbol. However, in practice, they were not completely orthogonal due to fractional timing issues. The issue of fractional timing error in the system will be discussed in greater detail in the section 4.3.2.1 Clock Drift. Another advantage to using slepians was that they were not lapped,

so the current message did not need to concern itself with calculating any overlap from the previous and next messages, making implementation much easier.

The slepians were designed using Matlab. The first step was to determine their bandwidth, and waveform time span. In the prototype network, the channel bandwidth was limited by the highest sampling rate of the codec, which is 48 kHz. At this sampling rate, the bandwidth of the pulse shaping waveforms had to be below 24 kHz. However, at higher bandwidths, the problems due to fractional sample offset increased. Thus, a compromise of 4 kHz bandwidth was reached. A 4 kHz bandwidth gave a total of 8 kHz, counting both positive and negative frequencies. This provided enough bandwidth to send data and pilots, and reduce fractional timing errors. The time duration of the waveform was determined next. This time duration corresponds to the length of a transmission. The only hardware requirement for this parameter was that it be an integral number of samples. However, for ADT to adapt to changes in a wireless channel existing in an indoor environment, it was estimated that the channel needed to be measured on the order of 10 ms [3]. Therefore, the time span for a waveform needed to be significantly less than 10 ms. It was decided that each waveform would be 128 samples. Sampling at 48 kHz, the block of slepians spanned 2.67 ms.

The raw slepians generated in Matlab were processed to make them useable. Only those slepians whose power concentrations in the specified band surpassed 99 % were used. This gave a set of real-valued waveforms, occupying –2 kHz to 2 kHz, centered around DC. The slepians were shifted up and down in frequency by 2.5 kHz and this also resulted in a guard bandwidth of 1 kHz centered around DC that allowed

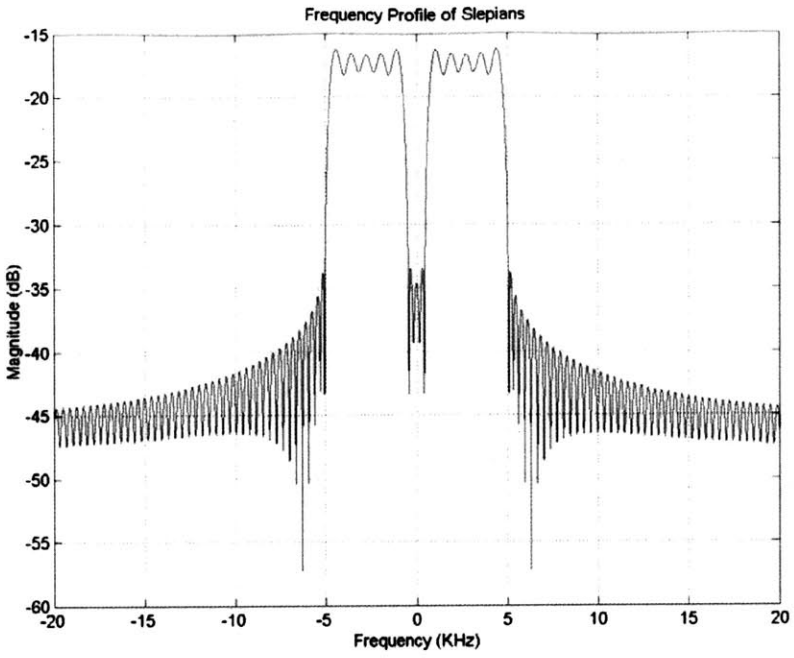


Figure 4.1 Magnitude plot of slepian frequency profile. The bandwidth of the slepians is 10 kHz with a guard bandwidth of 1 kHz around DC.

us to avoid dealing with DC offsets in the RF hardware. The resulting frequency profile is shown in Figure 4.1. All the low frequency components up to 0.5 kHz above and below DC have been removed.

The slepians were orthogonal in time, but needed to be sorted. Each slepian peaks at a time characterized by its eigenvalue. Thus, they could be sorted according to their eigenvalues. At this point, the slepians, were complex-valued waveforms, since they had been shifted in frequency. It was simpler to deal with real-valued waveforms, so the slepians were rotated in the complex plane to lie on the real axis. This is illustrated in

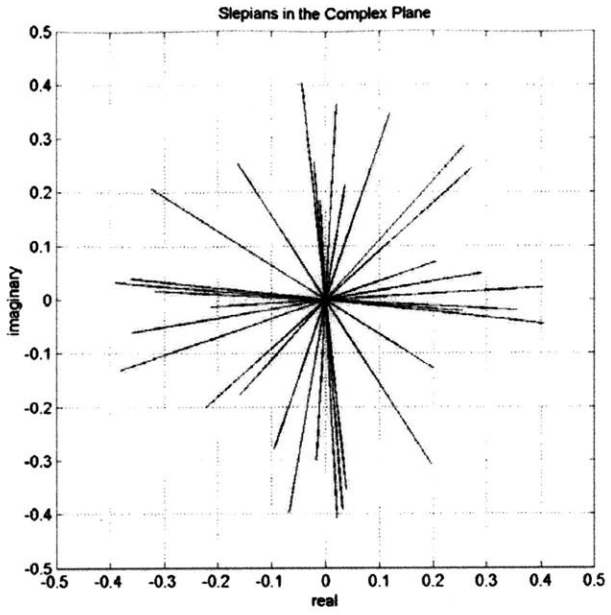


Figure 4.2 Plotting the slepians in the complex plane show that they are lines in the complex plane, so they can be rotated to lie on the real axis.

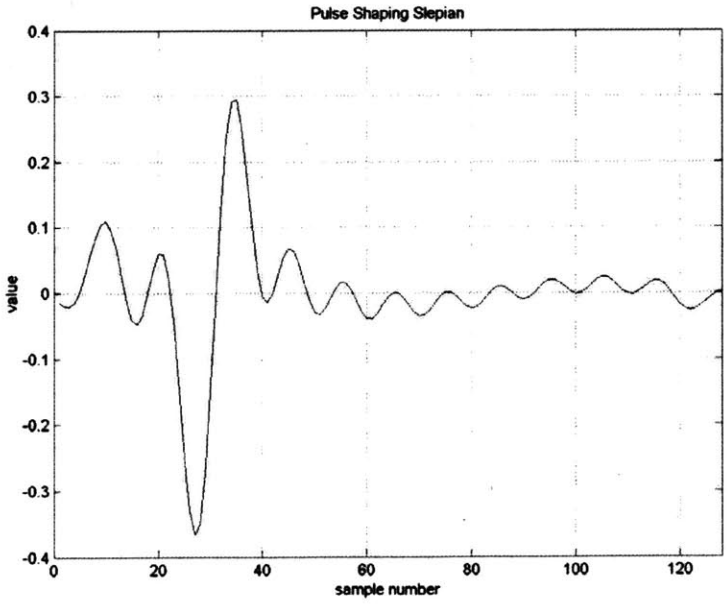


Figure 4.3 The fourth waveform from the set of slepians.

Figure 4.2. The rotation was done by multiplying by $e^{-i\theta}$, where θ is the angle the slepian formed with the real axis. The final set of pulse shaping waveforms contained a total of 18 waveforms. Thus each message to a mobile had 18 symbols. Since each message lasted for 2.67 ms, the total bandwidth was 6.75 kilosymbols/s, or 13.5 kb/s (each symbol carries two bits since we are using quadrature phase shift keying). Figure 4.3 shows the fourth slepian in the final set of pulse shaping waveforms.

4.1.2 Data Constellation

The data constellation was a Quadrature Phase Shift Keying (QPSK) constellation. Each symbol had a real and imaginary part. The four possible values were [1+*i*, 1-*i*, -1+*i*, -1-*i*], normalized to have magnitude 1, as plotted in Figure 4.4. Thus, each

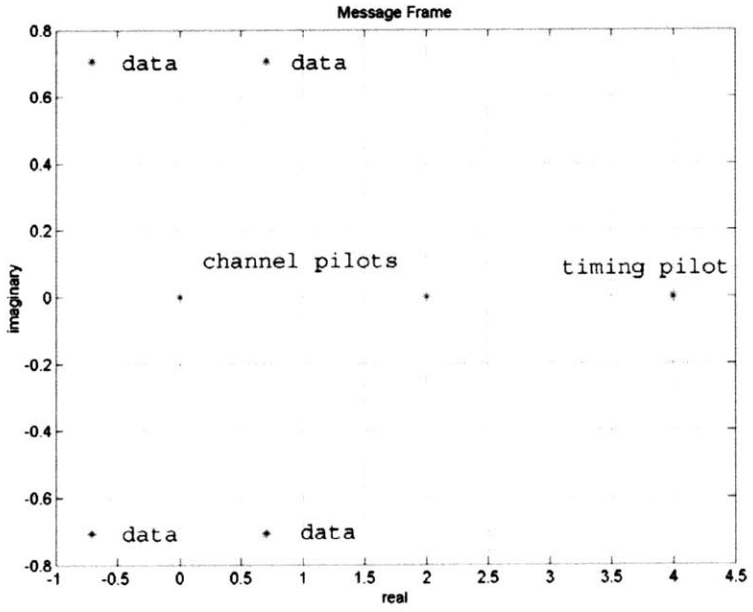


Figure 4.4 The data constellation and pilots plotted in the complex plane.

symbol represented two bits. A denser constellation was not used in order to reduce the bit error rate (BER). It has been shown that, in general, QPSK, a 4-symbol constellation, has a much lower BER than an n-ary constellation (such as n-QAM, n>4) for a given SNR [4].

4.1.3 Pilots

As discussed in section 3.2.2 Pilots, the only requirements on the pilots were that they be known values, and transmitted simultaneously. The channel estimation pilots, as mentioned, were chosen to be the identity matrix, with row one embedded at positions 1 and 3 in message one, and row two embedded at the same locations in message 2. They were given twice the magnitude as the data in order to increase their SNR, and thus their measurement accuracy. This is also illustrated in Figure 4.4.

The timing recovery pilot needed to have more power than the data and channel estimation pilots. Experimentation in Matlab determined that the power of the pilot needed to have at least nine times the power of the other signals, or three times the amplitude. A large power differential increased the security of acquiring timing. In the prototype, the power of the timing pilot symbol was 16 times the power of the data, and 4 times the power of the channel pilots. The timing pilot was placed in position 5 in each message. Figure 4.4 also shows the synchronization pilot.

The channel estimation pilots and timing pilot create some bandwidth overhead. The channel estimation pilots were sent in every data frame, occupying three data slots. Additionally, for better performance, the pilots were preceded and followed by blanking symbols ($0+0i$). Due to fractional timing offsets (discussed in 4.3.2.1 Clock Drift

section), the slepians were not entirely time orthogonal, so there was some power leakage between symbols. Blanking symbols were used to buffer power leakage. The blanking symbols were most necessary around the timing pilot since it was very high power, and would influence its neighboring symbols the most. However, they were also added around the measurement pilots as a simple way to improve performance. For a proof-of-concept of ADT, optimizing bandwidth utilization was not a primary concern. In summary, pilots alternated with banking symbols, sacrificing a total of seven symbols. The total possible downlink bandwidth was 18 symbols/frame, or 6.75 ksymbols/s. Thus, the pilots occupied a total bandwidth of 1.125 ksymbols/s, with the blanking symbols occupying a total of 1.5 ksymbols/s. Ultimately, 4.125 ksymbols/s, or 61.1% of the available bandwidth remained for data.

4.2 Base Station

The base station consists of a Transmit Module and a Kernel Module on the downlink side, and a Receive Module on the uplink side. The Transmit Module multiplexes the linear combination of messages onto the slepians, and sends this transmission over the wireless channel to the mobiles. It interfaces with the Kernel Module that provides the linear combination of messages. The Kernel Module also keeps the latest network matrix measurements, and its inverse that it uses to mix outgoing messages. The matrix measurements are received from the Receive Module, which continually listens for any incoming reports from the mobiles, and passes them onto the Kernel Module. Description will begin in the Kernel Module since this is where the messages originate.

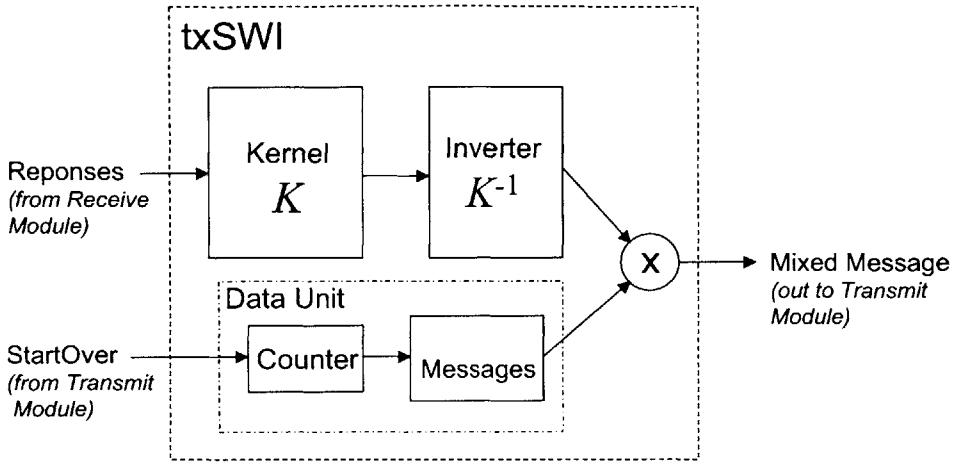


Figure 4.5 Block diagram of Kernel Module.

4.2.1 Kernel Module

The Kernel Module takes the data messages to be sent, and performs the mixing. The Kernel Module is implemented in a software thread (**txSWI**). The Kernel Module's four parts operate within this thread: the Data Unit, the Kernel, the Inverter, and the Mixer as shown in Figure 4.5. **txSWI** is invoked repeatedly to prepare new transmissions by the ISR triggered by the end of a DMA block transmission. A timing diagram for calling the Kernel Module (implemented in **txSWI**) is provided in Figure 4.6. **txSWI** runs at the start of the current message's transmission from the codec.

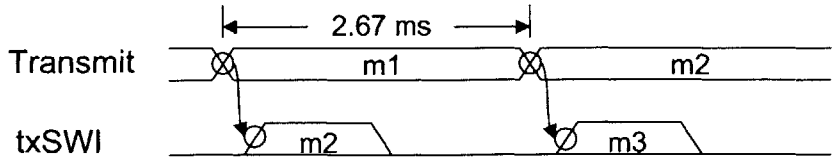


Figure 4.6 Timing diagram for Kernel Module (txSWI).

In **txSWI**, the Inverter unit runs first to create the mixing matrix. The Inverter checks the Kernel to see if new network matrix updates are available. If there are new updates, it inverts the network matrix, and passes the result to the Mixer. The Mixer also receives the messages to send from the Data Unit, and linearly combines the messages with the inverted network matrix. This result is then passed to the Transmit Module. Updating the Kernel occurs in a separate loop, discussed in section 4.2.3 Receive Module. The next few sections describe in detail the units of the Kernel Module.

4.2.1.1 Data Unit

Messages are supplied by the Data Unit. The Data Unit has a store of pre-computed messages to be sent to the mobiles. Each base station contains a copy of this message store. The message store contains 1000 message blocks to send to each mobile. The messages are composed of data symbols chosen at random from the constellation. Each message also has the timing synchronization pilot in the fifth symbol position, and all the surrounding blanking symbols. The stored messages are kept in quantized format. Figure 4.7 illustrates a sample quantized message. The message store is placed in the DSP's external memory bank, SDRAM0. The message store contains the messages to be sent to all the mobiles, in the order to be sent. For example, with two mobiles, the message store keeps the messages for each mobile interleaved with each other. The current messages to be transmitted are pointed to by a message counter. However, there is also a flag, **StartOver**, that redirects the message counter to the first message in the message store. This flag is used to synchronize the messages sent over all the base stations, and is set only on the first rising edge of the universal clock. Also, in order to

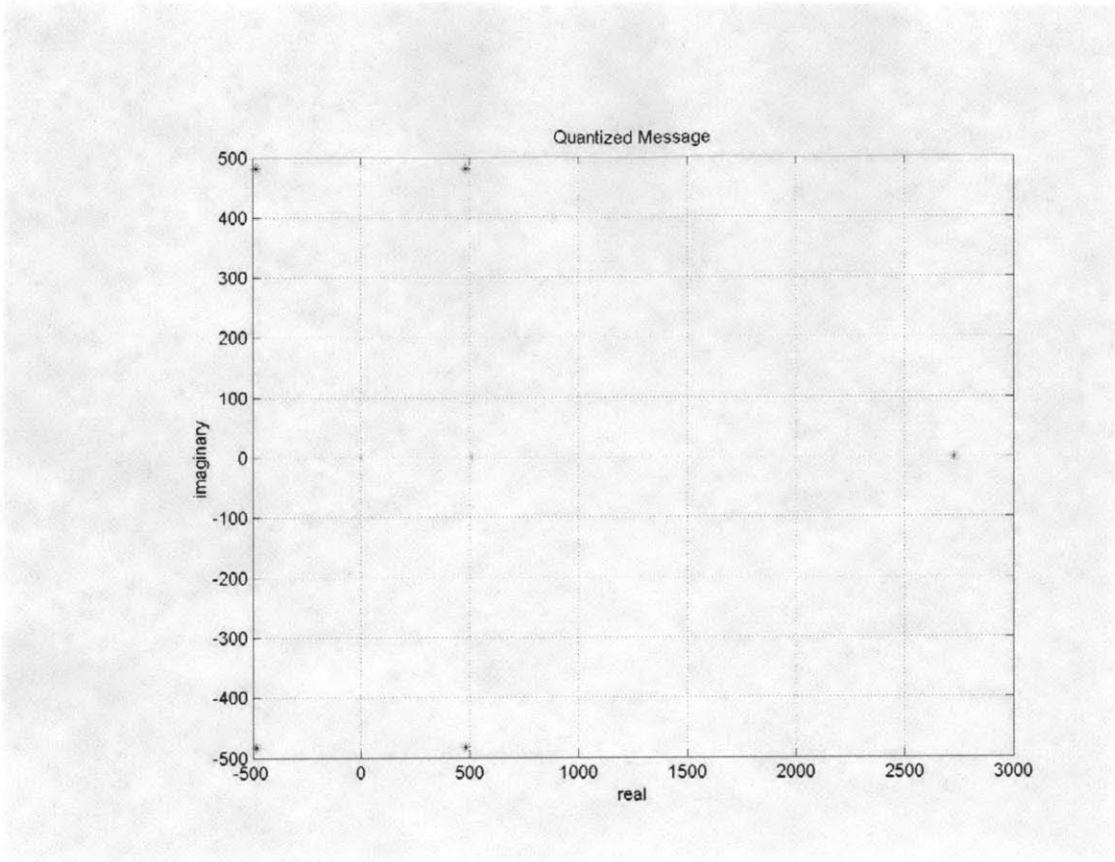


Figure 4.7 Message to mobile, quantized to signed 11 bit integers.

synchronize the logged data files between mobiles (for purposes of performance analysis), the data in the first fifty frames received by both mobiles is dynamically zeroed out.

4.2.1.2 Kernel Unit

The Kernel Unit holds the most up-to-date values of the network matrix. It is updated in a separate thread (**updateSWI**) which will be discussed in 4.2.3 Receive Module. In order to ensure that the Kernel Unit is not reading the memory space when **updateSWI** attempts to write to it, the Kernel Unit raises a flag, **busy**, that lets **updateSWI** know that it cannot write at that time.

4.2.1.3 Inverter Unit

The Inverter Unit receives the kernel from the Kernel Unit, and performs a two-by-two matrix inverse. The matrix elements are all complex numbers, so all operations must treat them as such. The inverse algorithm implements the procedures in Equations (1)-(3).

$$M = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{1}$$

$$M^{-1} = \frac{1}{\det(M)} \begin{pmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{pmatrix} \tag{2}$$

$$\det(M) = a_{11}a_{22} - a_{12}a_{21} \tag{3}$$

Notice that as the elements of M are complex, det(M) may also be complex. Thus, the division by det(M) follows the algorithm for dividing by complex numbers shown in Equation (4).

$$\frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{c^2+d^2} \tag{4}$$

Aside from this relatively straightforward formula, some complications arise from the fact that the system is a fixed-point system. That means the identity matrix $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ was represented as $\begin{pmatrix} 1024 & 0 \\ 0 & 1024 \end{pmatrix}$. The inverse of the latter, without the quantized

interpretation would be: $\begin{pmatrix} \frac{1}{1024} & 0 \\ 0 & \frac{1}{1024} \end{pmatrix}$ when the correct inverse is actually $\begin{pmatrix} 1024 & 0 \\ 0 & 1024 \end{pmatrix}$. To normalize for this effect, the calculated inverse of the network kernel is simply multiplied by 1024^2. For ease of implementation, the non-normalized inverse is calculated using the DSP's floating-point capabilities to preserve precision before converting back to the quantized format.

Another complication arising from fixed-point limitations as well as power limits occurs when the matrix inverse requires too much power. Saturation of either the codec or the RF hardware causes the mobiles to receive a corrupted transmission. To avoid this, the average power of the inverse is checked. If the average power is more than 6.25 times the average power in the unity matrix, then the dynamic range of the codec will be exhausted. If this is found to be the case, the inverse is first multiplied by a scale factor. This scale factor is found using the following equation:

$$scale = \sqrt{\frac{\langle I_{ij}^2 \rangle}{\langle L_{ij}^2 \rangle}} \tag{5}$$

where $\langle I_{ij}^2 \rangle$ is the power, or mean square of the elements in the identity matrix, and $\langle L_{ij}^2 \rangle$ is the power in the inverse of the measured network matrix. Since the scaling affects amplitude, the square-root was taken to arrive at the scale factor. In order to

preserve precision, the scaling is done as a floating-point operation. Note that changing the scale factor is essentially the same as a change in the channel, so the scale factor must be universal over all the base stations. The power in the inverse kernel matrix meets this requirement since the kernel is universal throughout the network. The scale factor must also not change too quickly on average, or else the ADT update frequency will not be able to adapt to the changes. Thus, a weighted average between the previous and current scale factor is taken.

The dynamic range of the system could have been improved by sacrificing the baseline SNR, however, this would have worsened the overall performance of the system. Moreover, it was found that for the lab environment, the range without scaling was sufficient for the average and good cases, although in the poor cases, scaling sacrificed SNR. This is because the power limitation algorithm takes effect when a mobile's signal reception is poor enough that it requires more transmit power than the system can give. Thus, the algorithm ends up limiting the power when it needs to be increased. However, the system still remains operational in this scenario, which is preferable to saturating the system, and sending corrupt transmissions. Furthermore, the proper action in that scenario would be for the end-user to simply move to a location of better reception.

4.2.1.4 Mixer

The Mixer takes the output of the Inverter Unit – the inverse of the network matrix (which may or may not have been scaled), and the output of the Data Unit – the two messages to be mixed. The Mixer then picks out the appropriate row in the inverse kernel (row one for base station one and row two for base station two) and performs a

matrix multiplication on the row and the two messages. The Mixer operates in terms of quantized values. Similar to the matrix inverse operation, the Mixer must account for the fact that the value '1' is represented by the value '1024.' Thus, the matrix multiplication divides the raw result by '1024.' The result is one message that is a linear combination of the two messages. This result is sent out of the Kernel Module to the Transmit Module.

4.2.2 The Transmit Module

The downlink side is completed at the Transmit Module. The Transmit Module is responsible for generating the wireless signals to the base stations, and synchronizing that transmission with other base stations. At the front end, it consists of an RF Transmitter, a Synchronizer and a Modulator. The RF Transmitter is described in 3.4.11 Transmit/Receive RF Modules. The Modulator receives the mixed messages from the

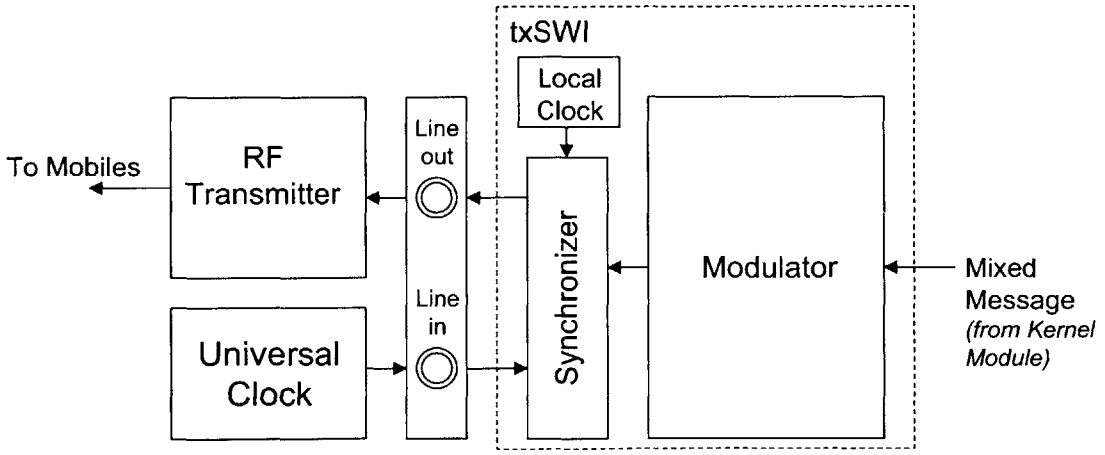


Figure 4.8 Block diagram of Transmit Module.

Kernel Module. It then modulates this onto the pulse shaping slepians. The resulting waveform is then passed through the Synchronizer, which synchronizes the transmission with the Universal Clock. This synchronized version is then transmitted over the wireless channel through the RF Transmitter as depicted in Figure 4.8.

4.2.2.1 Modulation Unit

Continuing the data path from the Kernel Module, the linear combination of messages is passed to the Modulation Unit. The Modulation unit is implemented within the same software thread as the Kernel Module (**txSWI**). This is because the Modulator only needs to run every time it is passed a new message from the Kernel Module. The Modulator takes this mixed message and modulates it onto the block of pulse shaping slepians, described in section of 4.1.1 Pulse Shaping Waveforms.

The slepians are stored in memory on the DSP board in the SBRAM section. The stored waveforms have all already been quantized as well. To modulate the messages onto the waveforms, the Modulator simply performs a matrix multiply between the message block and the slepians. The slepians are a 128 by 18 matrix, and the message is a vector with 18 elements. Thus, the result of the matrix multiplication is another vector that is 128 in length, as illustrated in Equation (6).

$$\begin{pmatrix} s_{1,1} & \cdots & s_{1,18} \\ \vdots & \ddots & \vdots \\ s_{128,1} & \cdots & s_{128,18} \end{pmatrix} * \begin{pmatrix} m_1 \\ \vdots \\ m_{18} \end{pmatrix} = \begin{pmatrix} t_1 \\ \vdots \\ \vdots \\ t_{128} \end{pmatrix} \tag{6}$$

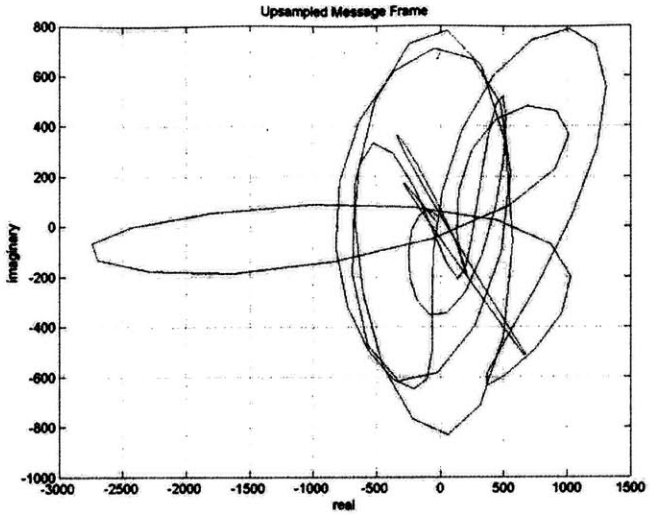


Figure 4.9 Plot of an up-sampled message frame in the complex plane. The curves show how the slepians are band-limited.

As with the Mixer, the Modulation Unit also works in quantized numbers. Figure 4.9 shows a quantized message in the complex plane after they have been up-sampled with the slepians. Notice that the waveform curves through the approximate locations of where the raw symbols lie. This is because the slepians are band-limited. If they were not, then they could abruptly jump from symbol to symbol.

4.2.2.2 The Synchronizer

The Synchronizer is responsible for synchronizing the transmissions from the base stations so that all the base stations begin transmitting the same messages at the same time. This method for accomplishing this task is somewhat non-standard, given the hardware limitations of the DSP evaluation boards. The evaluation boards provide crystal oscillators with the audio codecs. The crystals on each board have slightly different characteristics, which can change with temperature as well. Thus, if the codec

were programmed to sample at 48 kHz, the actual sampling rate might be slightly faster or slower than this value. This means a transmission of 128 samples from one base station would last a different length of time than the same 128 samples transmitted from the other base station, even if they began at exactly the same time. Over time, this difference would accumulate. For example, in a transmission block, the difference between the start times for the next transmissions would most likely be only a small fraction of a sample. After many blocks (several seconds) however, the difference between the start times would diverge to multiple whole samples. (Figure 4.10)

It was physically impossible to tie the codec clocks together in hardware. Furthermore, this would not have been a realistic situation. The more elegant solution, and the one implemented, was to have each base station keep track of the offset between their local sampling clock and a Universal Clock. This also took care of how to start the base stations at the same time. The base stations do not have to be started at the same time, but at the first Universal Clock edge, they start sending the same message number, synchronized to the Universal Clock edge. Thereafter, the base stations adjust the start of the transmissions by skipping or delaying a sample if their local clocks drift by a whole sample relative to the edge of the Universal Clock. The signal generator, described in section 3.4.12 Signal Generator, was the Universal Clock. The waveform was a 375 Hz

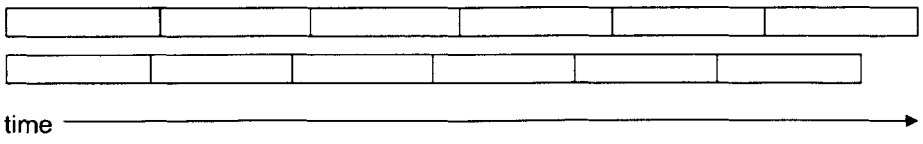


Figure 4.10 The difference in start times between the top blocks and the bottom blocks diverge over time if the blocks are not the same length.

square wave, with a peak-to-peak voltage of .5 V. The clock period was set to equal the duration of a message sent at exactly 48 kHz.

The Synchronizer implements the base station synchronization algorithm. The Synchronizer receives one block of transmission from the Modulator, and decides when to begin the transmission by looking at the local offset from the Universal Clock. Then it passes the transmission block to the RF Transmitter through the Line out interface. On the first Universal Clock edge, the Synchronizer resets the message number by asserting the **StartOver** flag to the Data Unit in the Kernel Module, described in. section 4.2.1.1 Data Unit. This takes care of the base station message synchronization.

In order to find the offset from the Universal Clock, the Synchronizer utilizes the audio codec, the serial port, and two DMA channels as described in the sample application in section 3.4.8 Using the DMA to Service the McBSP Serial Port and Codec. Figure 3.5 shows a block diagram of the hardware connections. For the base station's

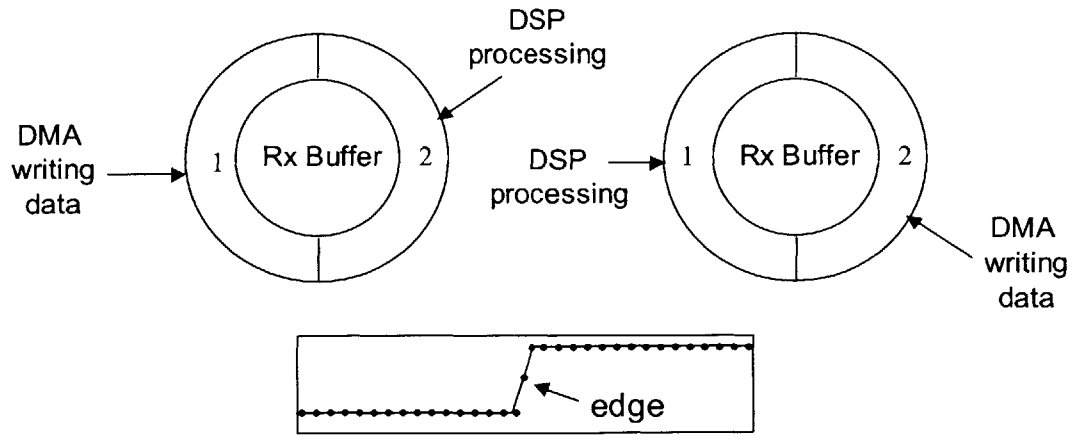


Figure 4.11 On each DMA input interrupt, the buffer the DMA was writing to becomes the buffer the DSP looks at, and the buffer the DSP was looking at becomes the new input buffer. The buffer of samples shows the rising clock edge, as sampled by the codec.

application, there are two buffers, each the length of a message. The input stream is the signal from the Universal Clock. This signal is sampled, and the DMA writes samples to buffer 1, while the DSP processes buffer 2. When the DMA interrupt is triggered, the buffers switch roles – the DMA writes samples to buffer 2, and the DSP processes buffer 1 as shown in Figure 4.11. When processing a buffer, the DSP simply looks for a very quick change in value. This quick change spans three samples: a minimum value, a middle value, and a maximum value. The 'edge' of the Universal Clock is the sample number of the middle value. This number is the offset from the Universal Clock edge, which is used to synchronize the output stream.

The output stream is also based on the DMA application, except that the DMA transfer length is half the size of a message. This splits the transmission of a message into two halves. During the transmission of the first half, the next message is placed into the output buffers. Adjusting the beginning of the transmission to the Universal Clock, may require starting the next transmission a few samples into the tail of the current transmission block. The amount of intrusion should be no more than a few samples. Since the DMA transfer is still on the first half of the block when the Synchronizer writes the new block, the DSP is guaranteed to not write to the same piece of memory that the DMA is reading.

As stated above, the offset from the Universal Clock edge dictates the number of samples to overwrite or skip in the circular output buffer. When the Synchronizer takes the modulated message from the Modulation Unit, it writes the samples into the circular output buffer, starting at the position pointed to by a WritePointer. On start up, the

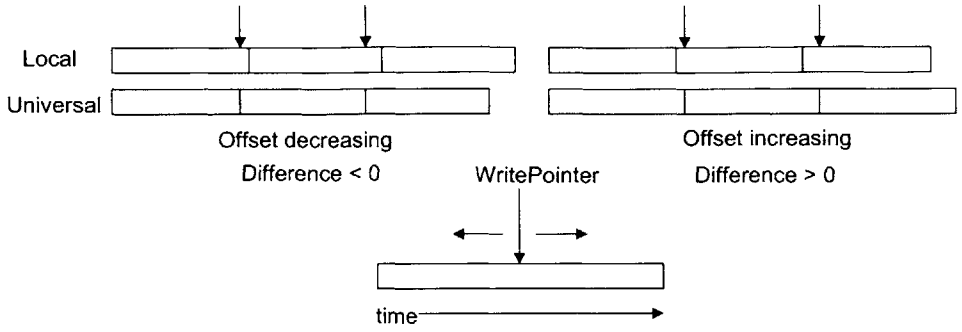


Figure 4.12 If the Local Clock is slower, the difference will be negative, so adding difference + WritePointer will cause WritePointer to move left (back) in time. If the Local Clock is faster, the difference is positive, so adding difference+WritePointer will cause WritePointer to move right (forward) in time.

output DMA begins transmitting from sample 0 of the circular buffer, and sets WritePointer to sample 128. Before the next block is written, the offset from the Universal clock is added to WritePointer, and the samples are copied into the buffer, starting from WritePointer. Then WritePointer is simply incremented by 128. The offset value is recorded. On the subsequent rounds, the *difference* between the old offset and the new offset (from the Universal clock) is added to WritePointer. If there is no difference, then WritePointer is not modified, and writing begins where it left off. If the difference is *positive*, WritePointer is incremented. A positive difference means that the local clock is too fast, so the new message should start later. Conversely, if the difference is *negative*, WritePointer is decremented. A negative difference means that the local clock is too slow, so the new message should start earlier. Adding this difference in offset to WritePointer neatly performs just that. Figure 4.12 illustrates this synchronization algorithm. Since the output buffer is steadily played through by the codec, this keeps the base station transmissions synchronized to the Universal Clock to

the nearest whole sample. This transmission is then passed to the RF Transmitter, which completes the base station's output data path.

4.2.3 Receive Module

The Receive Module's job is to listen on the uplink for the channel measurement replies from the mobiles, and to ensure that all the base stations have a unified copy of the network matrix. The network medium used is a standard Ethernet LAN. Note that in the prototype system, the uplink is a broadcast network, meaning that the mobiles broadcast the replies to all the base stations.

The Receive module consists of a Host Application running on the PC, and the Update Module. The Host Application runs a server to which the mobiles connect. When messages are received, the Host Application communicates the data to the Update Module through the HPI. The Update Module then updates the Kernel Module.

4.2.3.1 Host Application

The Host Application is a C++ Console Application running on the host PC. The application consists of two parts. One part is the server that listens on the LAN for any incoming connections. It continually listens for incoming connections on port 50, so multiple mobiles can connect to it at any time. Mobiles send messages to the server in a unique format containing a mobile ID number, and their channel measurements. The channel measurements from one mobile correspond to one row in the network matrix. Mobile ID numbers are unique to each mobile, and in a 2x2 network, the set of mobile ID's is $\{0,1\}$. The ID number tells the server which row the measurements are for. For

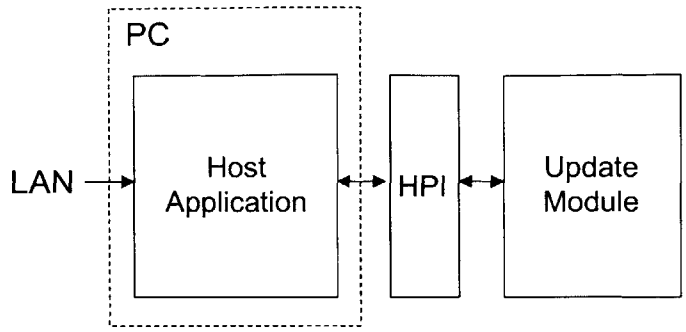


Figure 4.13 Block Diagram of Receive Module.

example, data from mobile ID 0 updates matrix row one, and data from mobile ID 1 updates matrix row two.

The second part of the Host Application is the HPI agent that transmits the responses to the DSP. The HPI port allows the HPI agent to have the capability of directly writing to DSP memory. When it is able to, the HPI port copies the responses to the response register in DSP memory. This response register is a dedicated portion of DSP memory that is hard-coded on both the DSP side and Host Side.

These two parts communicate through a flag, **pcDAV**. **pcDAV** is set when the server side receives responses from both mobiles. The server uses an internal counter to keep track of who has sent messages. If it has only received messages from one mobile, then the value of the counter is 1. When the counter reaches 2 (signaling that both mobiles have replied) the counter is cleared and **pcDAV** is set. The HPI agent checks **pcDAV** to see if there is a complete set of data to write. If there is, it writes it to the Update Module and then resets **pcDAV**. If there is no data to write, it waits until the next time around. The handshake between the HPI and the Update Module will be discussed in 4.2.3.3 Host PC to DSP Handshaking.

4.2.3.2 Update Module

The update module runs inside the software thread **updateSWI** on base station's DSP. The update module currently checks if there is an update to the matrix every time a message is transmitted. The frequency of checking is adjustable through a flag, **update**. A new update arrives from the Host Application approximately every 3-5 frames, or every 8-13.3 ms. The timing is inexact due to the nature of TCP/IP traffic, but the necessity for a stringent update cycle was found to be unnecessary. Before updating the kernel, the Update module performs a few checks. First, it checks if the Kernel Module is currently using the kernel. It does this by seeing if the variable, **busy** is set. If it is, then it causes the program to exit with an error, since in normal operation, this condition should never occur. If **busy** is not set, then the Update Module goes ahead and updates the kernel. The update involves a weighted average between the previous network matrix value and the current value. Experimentation showed that a weight of 80/20 between the old and new value works well.

4.2.3.3 Host PC to DSP Handshaking

This section describes the data synchronization between the Host Application's HPI Agent and the DSP's Update Module. Because the HPI agent is driven by the Host Application, and can write DSP memory directly, handshaking is needed to ensure that only valid data is used by the kernel module. The handshaking signal (**DAV**) is communicated through a dedicated register in the DSP's memory. **DAV** is high (1) when the Host Application has written new data to the response registers, and the Update

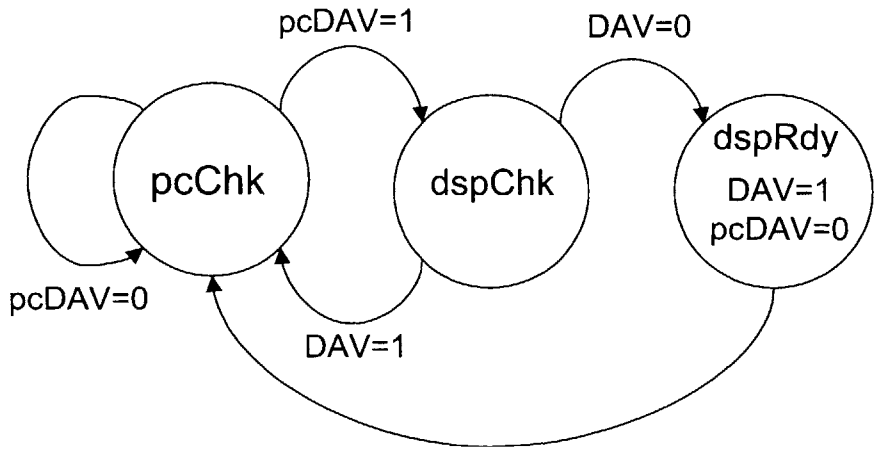


Figure 4.14 FSM for handshake between the DSP and Host on the Host side.

Module has not used it yet. The Update Module sets **DAV** low (0) after it copies the response register to the kernel variable. Thus, if **DAV** is high when the Host Application wants to write data, it must wait. If **DAV** is low when the Update Module wants to update, this means that no new data has been written, so it need not update the Kernel Module. The FSM (Finite State Machine) in Figure 4.14 illustrates the handshake between the Host Application and Update Module on the Host side. The Host Application continually checks its **pcDAV** flag. If it is set, then replies have been received from both mobiles, so the host moves to the dspChk state. There, it looks to see if **DAV** is set. If it is, then the Update Module is not ready, so the host goes back to checking **pcDAV**. Otherwise, the host continues onto dspRdy, where it sends the update to the Update Module, sets **DAV**, and clears **pcDAV**.

The FSM in Figure 4.15 illustrates the handshake between the Host Application and the Update Module on the DSP side. The Update Module starts in UpdateChk, and

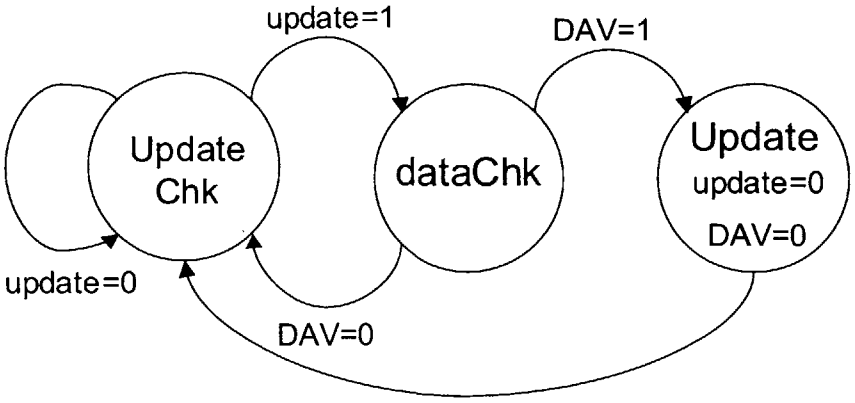


Figure 4.15 FSM for handshake between the DSP and Host on the DSP side.

checks if it is time to update the kernel. If it is, it moves to dataChk, where it checks if **DAV** is set. If it is not, then the Host has not sent data yet, so the Update Module returns to UpdateChk. If **DAV** is set, then it moves to the Update state, where it performs the handshake with the Kernel Module, and if permissible, updates the kernel. After this, the Update Module clears **DAV** and **update**.

4.3 Mobiles

The Mobiles consist of a Receive Module, a Decode Module, and a Reply Module. The Receive Module receives the wireless transmissions through the RF Receive Module, which demodulates the message from the carrier. The Receive Module then recovers timing with the base station transmissions. The recovered data transmission is then passed to the Decode Module. The Decode Module demodulates the data symbols from the pulse shaping slepians. The channel estimation pilots are then

read from the demodulated message by the Reply Module, and formatted into a message to send to the base stations.

4.3.1 Receive Module

The Receive Module receives incoming transmissions from the base station over the wireless network. The wireless transmission is demodulated down to baseband by the RF Receiver, and passed into the mobile through the audio line in. Since there is a large amount of attenuation between the receive antenna and the codec (this is done to avoid saturating the hardware mixer), the audio codec is programmed to compensate for this loss on the input. The receive module recovers timing between the base stations' transmission and the mobile's local receive frames. Recall that the transmission from the base stations arrive as an analog waveform (4.1 Wireless Message Protocol). The mobiles receive this analog signal, resampled at the same[1] sampling rate as it is played out by the base station. However, what appears to the mobile as the "first" sample in the message may in fact be the Nth sample in the transmission frame, where N ranges from 0 to 128. In order to decode the message properly, the mobiles must find the true start of the transmitted message. Knowing this, the Receive Module can construct a complete message to pass to the Decode Module.

The Receive Module receives the samples from the codec in the manner described in section 3.4.8 Using the DMA to Service the McBSP Serial Port and Codec. At a time, 128 samples (the length of a message) are received, and are stored in a triple buffer

[1] Although the base stations and mobiles are configured to sample at the same frequency, there is deviation from this frequency, which introduces additional error into the system.

system. While the DMA writes to one buffer, the DSP has access to the other two buffers. The most recent of these two buffers is used to calculate the timing offset from the base station, and the older of the two buffers is used to reconstruct one full message frame, in conjunction with the recent frame. This ensures that there is at least one message full message among the buffers. A fourth buffer of old received samples is also kept, in case the mobile's receive clock is slower than the transmit clock. If the mobile's receive clock is slow, then it will fall behind the transmission by a whole message. When this happens, two messages are decoded in one round, instead of one.

When the DMA signals (via interrupt) that current receive buffer is full, the buffers swap roles in a round robin fashion. The oldest buffer becomes the new receive buffer, the former receive buffer is used for timing recovery, and the buffer that was used last time for timing recovery is now used for message reconstruction. The last reconstruction buffer now becomes the spare buffer.

4.3.1.1 Timing Recovery Algorithm

As stated above, the first sample of the mobile's receive buffer may not be the first sample in the transmission. An example of this is illustrated in Figure 4.16. In this example, the mobile is 30 samples later than the base station. This means that the full message can be constructed by concatenating samples 0-97 of the buffer shown and samples 98-127 of the older buffer. The timing recovery algorithm finds this offset between the first sample of the mobile's receive buffer, and the start of the transmitted message.

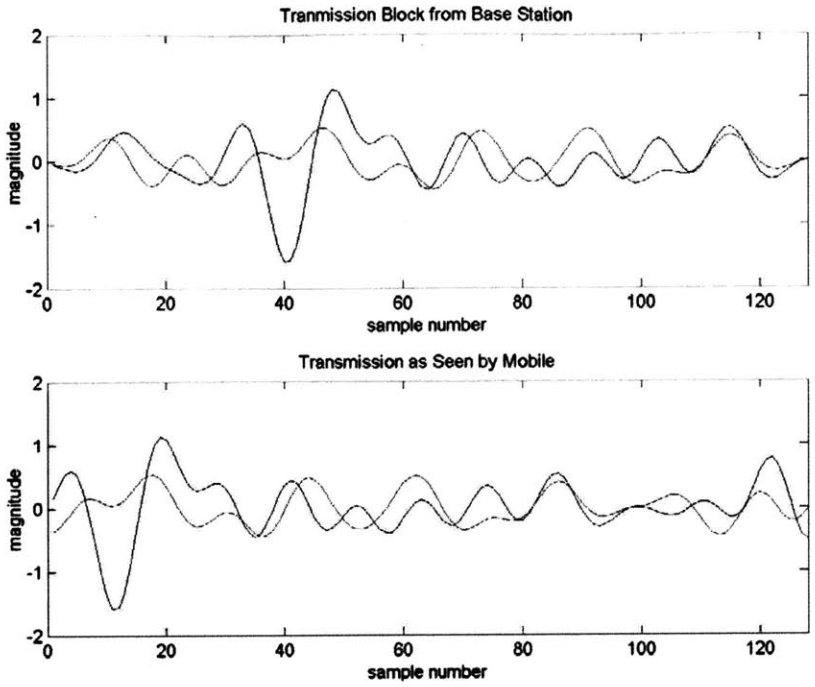


Figure 4.16 The top graph shows the I and Q channels of a full message, as sent by the base stations. The bottom graph shows the I and Q channels of the incoming receive buffer of the mobile. Using the large dip from the timing pilot, it is easy to see that the mobile's frame starts at sample number 30 in the base station's frame.

To find the offset, the mobile calculates a correlation between the latest received sample buffer, and the pulse-shaping waveform that the timing recovery pilot is modulated onto. The formula for the correlation is:

$$\sum_{N=0}^{127} x[n]s^*[n-N] \tag{7}$$

where s*[n] is the complex conjugate of the slepian and x[n] is the received waveform. When N equals the sample number in the mobile's frame where the transmission actually begins, the magnitude of the correlation will be maximized. As mentioned in section

4.1.3 Pilots, the power in the timing pilot had to be much greater than that of the random data modulated on the other slepians for the correlation to yield this offset. The mobile finds the sample number of the maximum magnitude of the correlation to find N. In the above example, N=97. This number is used to reconstruct the full message from the buffers of stored samples. Reconstructing a buffer occurs by taking samples N+1 to 127 of the older buffer, and appending samples 0 to N of the current buffer.

As discussed in section 4.2.2.2 The Synchronizer, the sample rate depends on the specific crystal that the audio codec utilizes. This means that the drift caused by a slight difference in sampling rate (and thus block length) affects the location of the offset. If the mobile samples faster than the base stations, then the offset will increase over time. Conversely, if it samples slower, the offset will decrease. Thus, it is necessary to recalculate the offset every transmission block. This drift in synchronization means it is periodically necessary to decode two messages in one run, or to skip decoding. The former case occurs when the mobile's receive clock is slower than the transmission (offset decreases), and it is in this case that the mobile uses the fourth buffer. The latter case occurs when the receive clock is faster.

4.3.1.2 Implementation

In practice, performing a correlation with the DSP is an expensive operation. It is much more efficient to work in the frequency domain, where the correlation operation turns into a simple vector multiplication. Thus, the implementation of the timing recovery algorithm is as follows. First, the Receive Module transforms both the newest receive buffer, and the timing pilot's slepian to the frequency domain, with a Fast Fourier

Transform (FFT). The complex conjugate of the FFT of the slepian is then multiplied to the FFT of the received signal. This vector product is the FFT of the correlation. It is transformed back into the time domain, and its absolute value is calculated. The sample number of the maximum of this vector is found, and returned as the desired offset. The equivalence between the algorithm and its implementation arise from the duality properties between the time domain and the Fourier domain [5].

4.3.1.3 Performance Improvements

The accuracy of the calculated offset is very critical. There was some power leakage between symbols caused by timing issues. This power leakage also caused noise on the peak of the correlation. This noise was enough to create whole sample errors in the synchronization offset calculation. If the offset is off by even one whole sample, the message will not be recognizable. A weighted average over the correlation function was employed to reduce the effect of the noise. Since the data is essentially random, averaging would reduce the noise. Also, the offset will drift as the mobile's sampling rate drifts in reference to that of the base station. The averaging should allow the offset to track this drift. In practice, a 20 % weight on each new correlation function was most effective at achieving these goals.

4.3.2 Decode Module

The Decode Module receives complete messages and demodulates them into data symbols, and channel measurements. The Decode Module was implemented in a SWI, **decodeSWI**. Inside this thread, the mobile simply takes the synchronized message and

multiples it with the transposed matrix of slepians. As in the base stations, the mobiles store these waveforms as an array in memory.

4.3.2.1 Clock Drift

Some error arises from the conversion of the incoming analog waveform to discrete samples. There is a certain amount of random noise from the codec itself. More substantial than this, however, is error from fractional sample offset. If the mobiles manage to sample at the same points in the waveform which the base stations used to construct the analog waveform, then there will be no fractional sample error. However, this condition is very unlikely. Even if initially the transmit and receive sampling clocks are in phase, slight differences in crystal oscillators mean that the codecs most likely are not sampling at exactly the same frequency. This causes a small phase difference over each frame that, over many frames, accumulates to a whole sample shift. The timing recovery algorithm tracks this drift over the whole sample increments, but cannot adjust to the fractional sample offsets. This fractional sample error shows up as noise in the decoded data. In order to alleviate the noise, the bandwidth of the pulse shaping waveforms was chosen to be 4 kHz. Slepians of 4 kHz bandwidth are relatively slowly varying in time, but still allow enough data to be sent per message. Figure 4.17 and Figure 4.18 show the noise added purely by fractional sample offset when no blanking

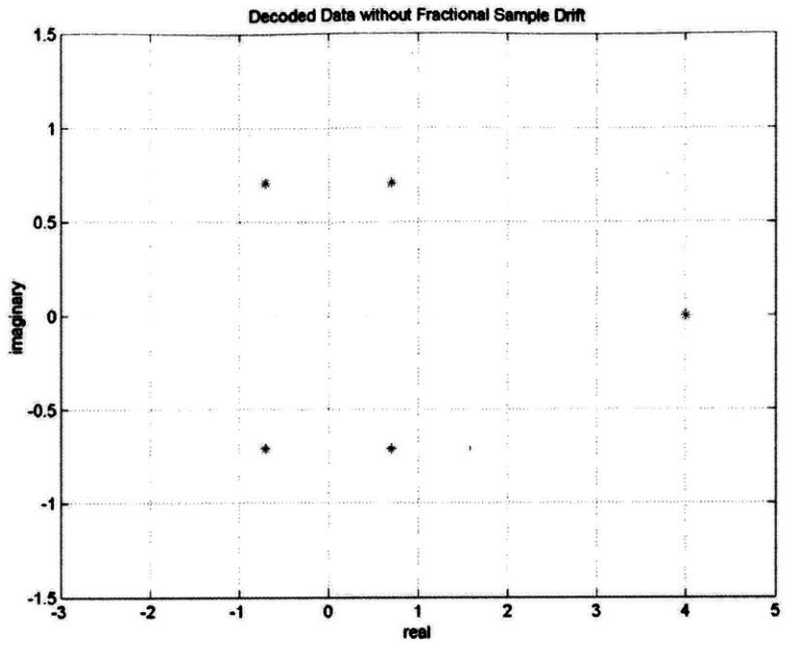


Figure 4.17 Decoded data where there is no fractional sample offset

symbols are used. These plots, produced by Matlab simulation, mirror the results seen on the real system[2].

The plots with the fractional sample offsets show two distinct characteristics. The first is the four lobe pattern centered around the actual symbol location. This four lobe pattern is best explained by the linear nature of the drift causing a changing fractional sample offset. Each symbol is represented by two bits, and each bit exhibits an

[2] To see only the effects of fractional sample offset in the system, results were observed by replacing the wireless network by an ideal network (wires), and the adaptation was turned off.

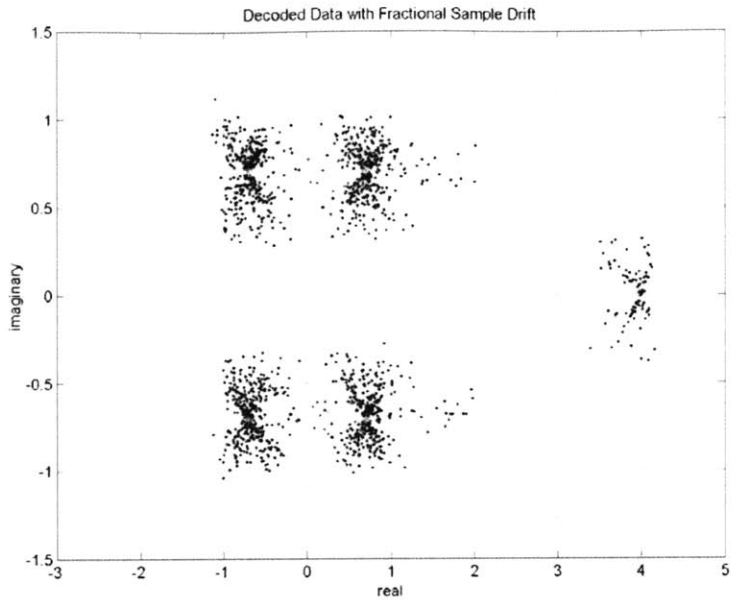


Figure 4.18 Decoding data with clock drift, without blanking symbols.

independent linear drift. The other characteristic is the horizontal streak of points, which appear to drift toward the right. These streaks are caused by a small amount of symbol leakage from the high power of the timing pilot. Adding in blanking symbols to isolate the timing pilot from the data successfully removed these horizontal streaks. Note that power leakage indicates that the slepians are not entirely orthogonal in time, though they are theoretically. This points to fractional offset between the sampled receive waveform as a cause for the loss of this property.

4.3.3 Reply Module

The Reply Module is responsible for formulating the message with the channel estimation pilots to send back to the base stations. It takes the demodulated message and picks out the symbols that are the channel estimation pilots. In this implementation, the

pilots are symbols one and three in the message frame. These symbols are then passed to the Host Application, which sends the message over the LAN to the Base Stations.

4.3.3.1 Message Unit

There are two basic parts to the Reply Module. The first is the Message Unit, which creates the response with the channel estimation pilots. These responses have two purposes. The primary purpose is to send the channel measurements to the base stations. However, in the prototype, these messages were also used to send the entire decoded message to a display server, which displayed the decoded messages in real time, and logged them for analysis. The display server was kept separate from the system because the base station and mobile PCs were already heavily loaded from running the uplink code. The message created by the Message Unit thus contained the row of channel measurements received by that mobile, the data, and a unique mobile ID, one for each mobile.

The Message Unit was implemented inside **replySWI**. After the Decode Unit finishes demodulating the symbols from the pulse shaping waveforms, **replySWI** is posted, notifying the Message Unit of new channel measurements. At this point, the Message Unit constructs the message, and attempts to send it to the Host Application. If the attempt is unsuccessful, the Message Unit buffers the message. On the next successful attempt, all buffered messages are sent. The Message Unit has the capability of buffering up to 20 messages.

4.3.3.2 The Host Application

Like the base station, the mobiles have a Host Application simultaneously running on the host PC. This application has two modules: the HPI Agent, that reads the messages from the Message Unit, and the Client, that sends them out to the base stations and display through the LAN. The HPI Agent communication with the Message Unit using the HPI over the PCI bus. In order to do this properly, there is a handshake performed between the HPI Agent and the Message Unit. This handshake is described in more detail in section 4.3.3.3 DSP to Host PC Handshaking.

The HPI Agent and the Client also communicate internally via handshake as well. The HPI Agent is continually reading the messages from the Message Unit, whenever they are available. The HPI Agent notifies the Client that messages are available to send using the flag **newData**. This flag is cleared by the Client when the messages have been sent to all the servers. If the HPI Agent receives new data from the Message Unit before the Client clears **newData,** the HPI Agent appends the message to the end of the buffer. If **newData** is cleared, then the old messages are cleared and buffering begins anew. The HPI Agent has the capability of buffering up to 100 messages. If this is not enough, the Host Application exits, since this is indicative of a fatal error, such as a disconnected socket.

The Client continually checks **newData**, as well as the state of the sockets. Each Client of a mobile has three connections to monitor. One to each of the base stations, and one to the corresponding display. When the flag **newData** is raised, the Client sends the data on any socket that is available. If the socket is a base station socket, only the most

recent message is sent. However, if the socket written to is the Display server socket, then all the buffered data is sent. After all the sockets have been serviced, the Client clears **newData**.

One small complication that arises from using sockets for the uplink is the fact that sockets are a high level abstraction over the lower level network protocol TCP. TCP has provisions for efficient, reliable transportation of messages, which make it somewhat tricky to work with for real-time control applications, such as ADT. Window's sockets implementation of TCP streams have buffering. When the Host Application writes to an outgoing socket, the data gets written to a transmit buffer. However, this data may not be sent out on the network immediately. There are several checks that TCP does to see if it can satisfactorily send data. If the window of unacknowledged packets is 0, then the data cannot be sent. The window is not a problem for this particular application because the LAN is a high-speed network, and the data rate generated by the mobile is low. Another check that TCP does before sending a packet is known as Nagle's Algorithm. This is simply an algorithm to buffer small sends into larger sends, or until 200 ms has passed, alleviating the overhead of a TCP packet. For a real-time control application such as ADT, the 200 ms delay is intolerable. Thus, the solution is to disable Nagle's Algorithm. This means that if it is possible to send the updates, the Client will do so immediately. This timing, though not exact due to the reasons described above, occurs on average every 3 to 5 blocks (8-13.3 ms). This is within the range recommended by the initial estimates of ADT. Furthermore, buffering on the DSP and Host Application ensure that reasonable network delays do not cause lost data in the logs.

4.3.3.3 DSP to Host PC Handshaking

The Message Unit and the Host Application's HPI Agent have a handshaking protocol to synchronize data transfer. On the DSP side, the handshake occurs each time **replySWI** runs. The flag **DAV** is used to indicate the number of bytes the Message Unit has in the reply buffer. Each time a new message is decoded, the Message Unit enters the DAVchk state, and creates a reply. Then it checks **DAV**. If **DAV** is less than the maximum reply buffer size (20 frames * 76 bytes/frame=1520 bytes), then the Message Unit enters the buffer state, where it appends the message to the end of the reply buffer. Then it increments **DAV** by the size of the reply (76 bytes). If **DAV** is already the maximum size, the Message Unit exits with an error, since the DSP should be able to send replies with enough frequency that a backlog this large should not occur. This algorithm is illustrated in the FSM in Figure 4.19.

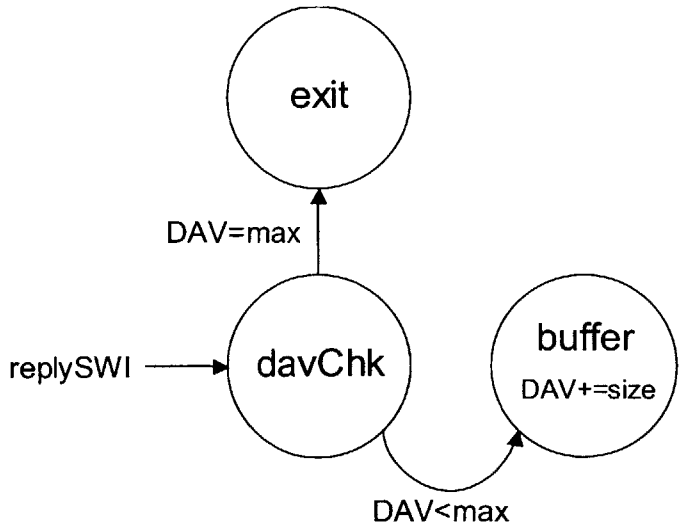


Figure 4.19 FSM for the handshake on the DSP side.

On the side of the Host, the handshake works as illustrated in Figure 4.20. The HPI Agent begins in davChk, where it constantly checks **DAV**. If **DAV** is greater than 0, then it moves to bufChk, where it sees if it has room to copy the data to its buffer. The location to copy to is determined by **newData**. If **newData** is not set, then the data overwrites old data starting from the front. Otherwise, the host checks if there is enough room to append the messages to the end of the buffer. If there is no space, the program exits with an error since this is indicative of a problem such as a network error. If there is enough room, which is the normal case, the Host copies the number of bytes specified by **DAV** out of the shared reply buffer, sets **DAV** to 0, and **newData** to 1. Then, the HPI Agent returns to its start state. Setting **newData** to 1 notifies the Client of new data to send over the network. The HPI Agent will buffer all the messages from the DSP until

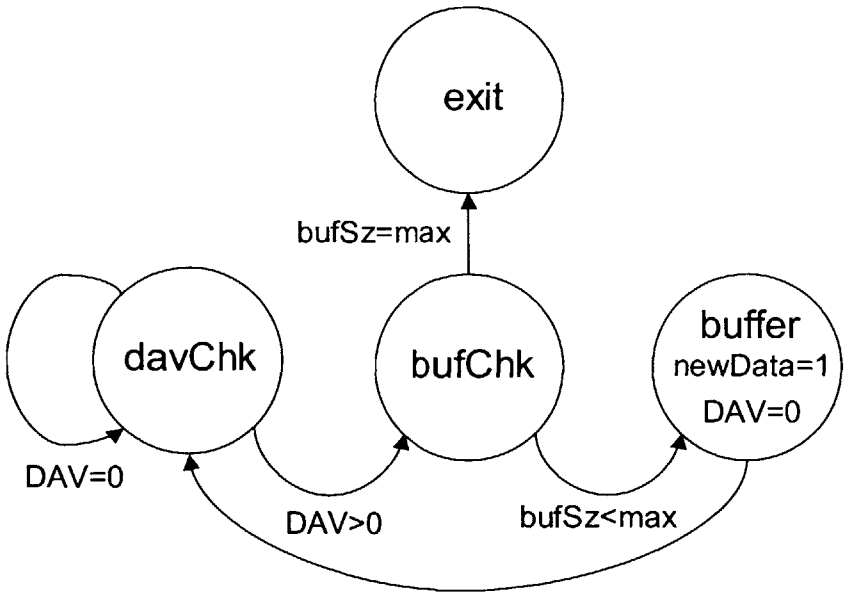


Figure 4.20 FSM for the handshake on the Host side.

the Client sends them. This was to ensure that the data logged by the display server was complete. The base stations only need the most recent channel measurements.

4.4 References

1. D. Slepian and H.O. Pollak, "Prolate Spheroidal Wave Functions, Fourier Analysis and Uncertainty – I," *Bell System Technical Journal*, **Jan.**, 43 (1961).

2. D. Slepian, "On Bandwidth," *Proc. IEEE*, **64**, 292 (1976).

3. B. Shraiman, M. R. Andrews, and A. Sengupta, Technical Report No. ITD-00-40485E, Bell Labs, Lucent Technologies (unpublished).

4. M. S. Roden, *Analog and Digital Communication Systems* (3rd ed., Prentice Hall, Englewood Cliffs, NJ 1991).

5. C. T. Chen, *System and Signal Analysis* (Holt, Rinehart and Winston, Inc., New York, NY 1989).

Chapter 5

Experimental Results

5.1 Experimental Environment

The prototype system was tested in the indoor environment of a large laboratory in Bell Labs. The room was approximately 30 feet by 20 feet in area, and 15 feet high, and there was one door to enter the room. The room also had one wall of outside facing windows, with tall trees in front. The room contained many items, including lab equipment and office equipment. The noise power in the room itself was measured to be about –35 dBm. Several runs were made, with various channel conditions. Several runs were made with very little movement in the room, while others were made with lots of movement varying the channels; the latter channel characteristic was created by moving a metal plate quickly through the wireless network.

5.2 Results

Performance of the system was measured in terms of bit error rate (BER) versus signal-to-noise ratio (SNR). The BER was measured by comparing the symbols sent to the symbols received. Received symbols were parsed from the raw decoded data using simple zero decision lines. Table 5.1 shows the decision algorithm for decoding the raw symbols. Figure 5.1 shows raw decoded symbols plotted in the complex plane.

Table 5.1 The decision algorithm for parsing decoded data into data symbols.

Real	Imaginary	Result
≥0	≥0	$1+i$
≥0	<0	$1-i$
<0	≥0	$-1+i$
<0	<0	$-1-i$

The SNR was calculated by measuring the noise of the raw decoded data values, compared to the signal power. The decoded data, when plotted in the complex plane, formed a QPSK constellation, where each symbol region created a cloud, centered around the expected symbol value (see Figure 5.1). The difference between each decoded symbol and the expected symbol value formed a vector in the complex plane. The

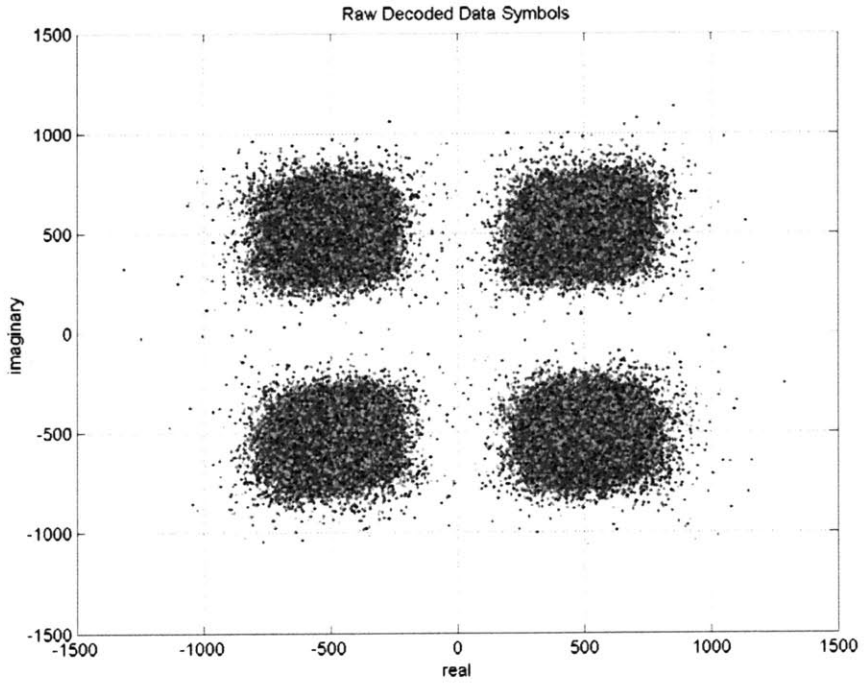


Figure 5.1 Raw decoded data symbols over 40,000 frames, plotted in the complex plane.

squared magnitude of this difference vector, averaged over many frames, compared to the squared magnitude of the expected symbol thus gave the average SNR. Specifically, the SNR was:

$$SNR \quad (dB) = 10 * \log_{10}\left(\frac{\mathbf{x}^2}{\left\langle |\mathbf{x}-\mathbf{y}|^2 \right\rangle}\right)^{\frac{1}{2}} \tag{1}$$

where **x** was the expected value of the data point, **y** was the actual value of the data point, and **x** and **y** are complex. $\left\langle |\mathbf{x}-\mathbf{y}|^2 \right\rangle$ was the mean squared error between the decoded symbol and the expected symbol, averaged over all the data decoded in the run. The square root of this ratio was taken, and the result gave the *SNR of signal amplitude versus noise amplitude, rather than the ratio of their powers*. The calculated SNR values and their corresponding BERs are shown in Table 5.2. They have also been plotted against a theoretical graph for the expected performance of QPSK in similar conditions. The graph in Figure 5.2 shows that the observed performance of the system is close to the theoretical performance for QPSK. The theoretical QPSK curve was extracted from the

Table 5.2 Table of BER versus SNR for the mobiles in various run conditions.

Mobile No.	Run No.	Condition	SNR (dB)	BER
0	2	no move	5.89	2.40×10^{-3}
0	4	Move	6.16	8.22×10^{-4}
1	4	Move	7.23	4.16×10^{-4}
0	3	no move	7.25	4.59×10^{-4}
1	3	no move	7.46	3.26×10^{-4}
1	2	no move	7.57	2.09×10^{-4}
0	1	no move	7.61	1.45×10^{-4}
1	1	no move	7.88	1.35×10^{-4}

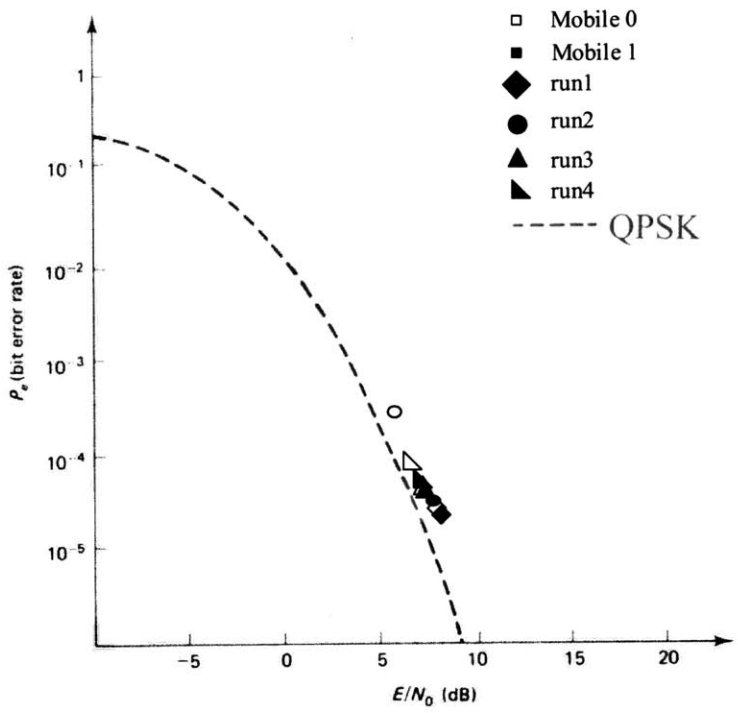


Figure 5.2 Plot of system BER versus SNR viewed alongside the theoretical performance of QPSK.

graph on p. 468 of Ref 1. The results for both mobiles are shown for each run.

Notice that in run 2, mobile 0 performed significantly worse than mobile 1. On this run, the display showed that amplitude of the values for the decoded constellation were reduced. This indicated that the scaling algorithm was reducing the power output of the transmitters in response to a mobile requesting more power than the system could output. From these results in Figure 5.2, it can be seen that this condition was caused by mobile 0. Mobile 0 was receiving poor signal quality, as shown by its poor average SNR, and resulting BER over the run. Its low pilot measurement caused elements in the inverse of the network matrix to be large enough to cause the power limitation algorithm

to take effect. Since this scaling was done uniformly over the network, the amplitude of the signal to mobile 1 also decreased by the scale factor.

However, since the signal quality to mobile 1 was good, the SNR remained high, and the resulting BER low, resulting in the same performance observed in good conditions. This is an important observation, namely that poor performance of one mobile need not affect the performance of other mobiles.

For run 4, a large metal plate was moved quickly through the network, and the lab door, which was situated near the transmit antennas, was opened and shut several times during the run. Additionally, near the end of the run, the antenna for mobile 0 was moved, then returned to its original position. Before moving the antenna, the performance of the mobiles over this run was basically the same as the runs with no movement, and very little activity in the room. The difference in SNR and BER between run 4 and run 1 were the result of different base line environmental conditions, since the runs occurred on separate days, and over different frequencies.[1] The conditions of run 3 were more similar to that of run 4 since the experiments were run in the same day, without changing the wireless frequency. The difference was that in run 3, there was very little movement in the room. As shown in Figure 5.2, the performance of ADT when metal objects in the environment are changing quickly is largely unaffected. This indicates that the frequency of feedback, which was on the order of every 10 ms, was sufficient to handling slow changes, such as those caused by human movement.

[1] Before an experimental run, the wireless network's frequency often needed to be changed because of external noise on certain frequencies. The experiment was designed to run over frequencies in the range of 868 kHz-890 kHz.

Toward the end of run 4, an antenna was moved to see how this affected performance. In the recorded data for both mobiles, this event is marked by a steep dip in SNR, and a marked increase in the total number of errors over those frames. This is shown in Figure 5.3-Figure 5.6. Results from run 2 demonstrated that one bad channel did not affect the other channel. However, in run 4, both mobiles were affected. The difference between the two cases can be seen by looking at the condition values of the network matrix. The condition of the network matrix is simply the ratio of the largest eigenvalue to the smallest eigenvalue. If the condition number is very large, then the matrix is nearly singular. For run 2, the values for the condition of the network matrix are plotted against the frame number in Figure 5.7. Its maximum condition value is 6.3007, and the mean condition value is 1.9086. In run 4, the condition of the network

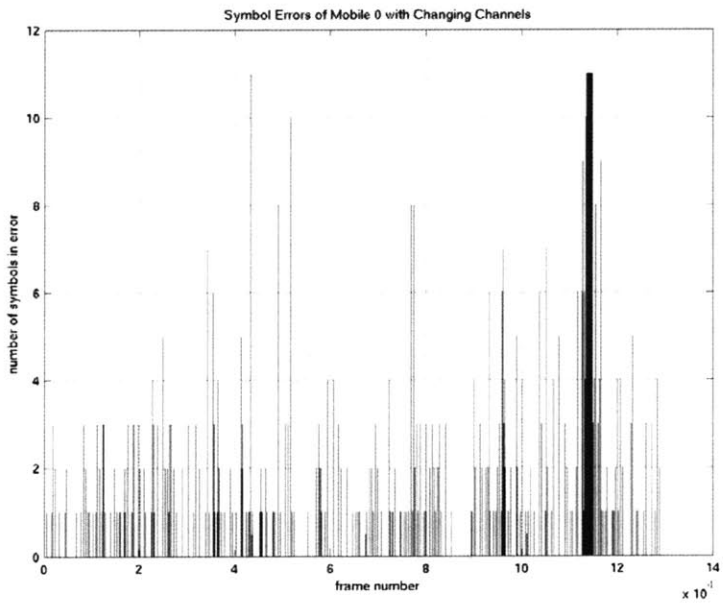


Figure 5.3 Number of Errors per frame for run 4 of mobile 0.

matrix is plotted in Figure 5.8. Here, the maximum value of the condition value was 408.1, and it occurs when the antenna was moved to the null position. Excluding this instant in time, the average of the condition values was 1.49. This indicates that, as expected, having a non-singular network matrix is key to the performance of ADT. Note that as the number of base stations increase, the probability of a mobile not being able to receive signal from any base station decreases exponentially. This is roughly because there is some probability, $P < 1$, that a mobile cannot receive signal from a base station. This probability is independent for each base station[2]. Thus, the probability that the mobile cannot hear any base station well is P^N, where N is the number of base stations [2].

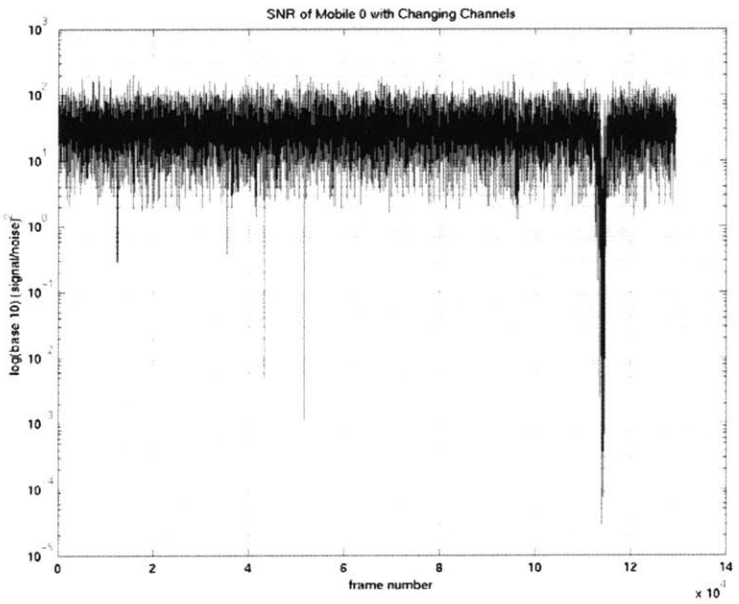


Figure 5.4 SNR per frame of run 4 for mobile 0.

[2] The probabilities are independent in standard scattering environments, where the base stations are well spaced and scattering occurs because of objects such as trees, buildings, and people.

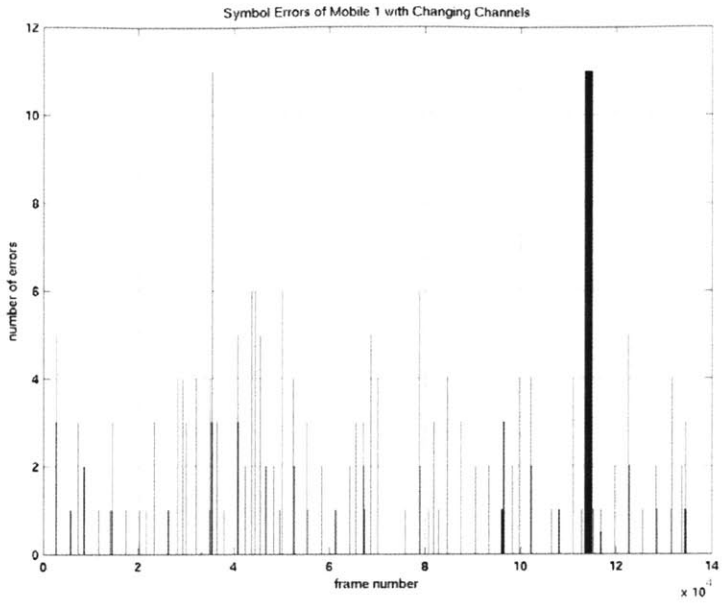


Figure 5.5 Number of errors per frame for run 4 of mobile 1.

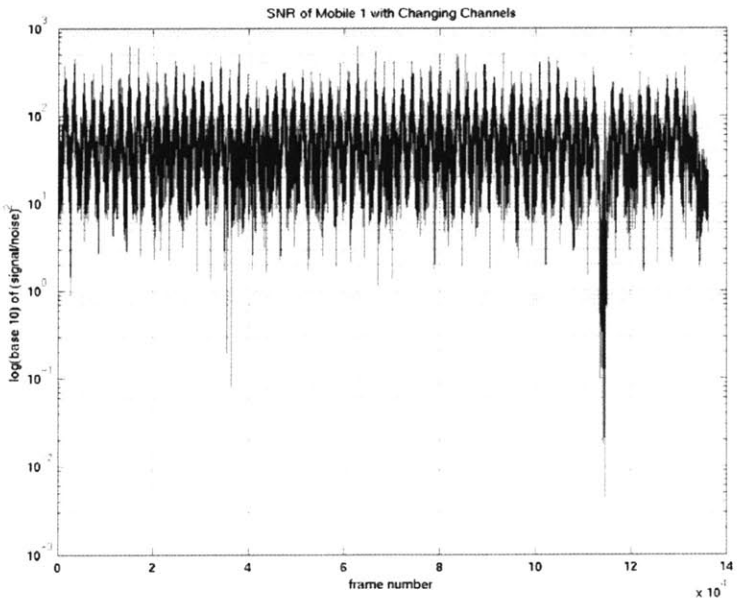


Figure 5.6 SNR per frame for run 4 of mobile 1.

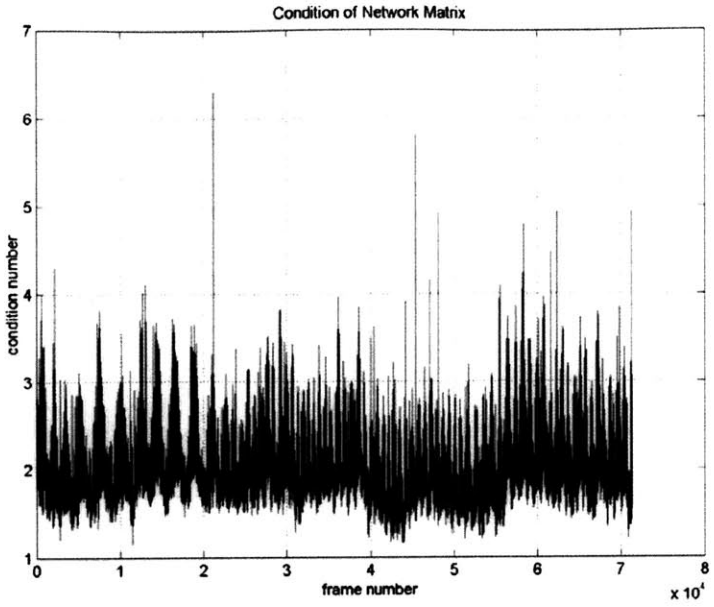


Figure 5.7 Condition values of the network matrix for run 2.

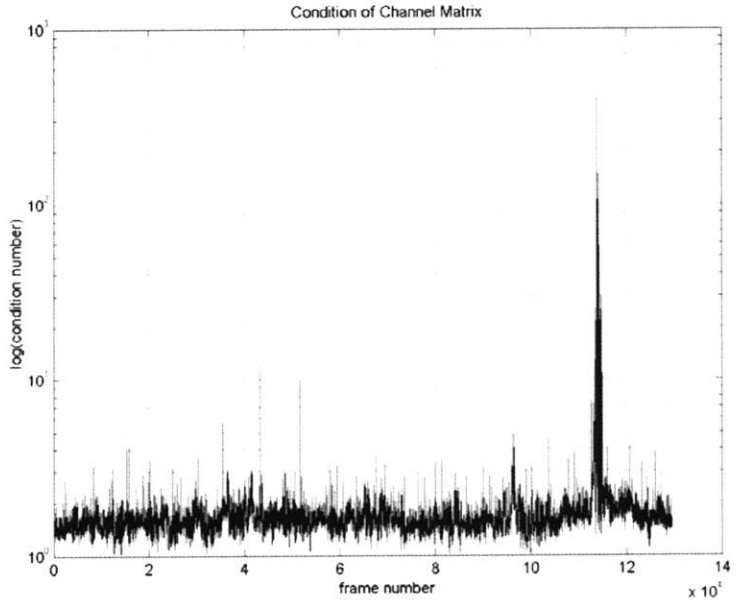


Figure 5.8 Condition values of the network matrix for run 4.

As discussed in section 4.3.2.1 Clock Drift, much of the noise in the data constellations was attributed to fractional sample offset. The graphs in Figure 5.9 and Figure 5.10 shows the SNR for mobile 0 and mobile 1 against frame number. There are a few important characteristics to notice. The first is the cyclic nature of the noise. For mobile 0, the period is about 650 frames, and mobile 1, the period is about 2200 frames. These periodic cycles in the SNR coincide with the relative drift of each mobile to the base station clocks. The codec crystal on mobile 0 is faster in relation to the universal clock, requiring about 650 frames to drift one whole sample. Mobile 1, is also faster, but has a much slower rate of drift. It takes about 2200 frames to drift a whole sample. The relative drift rates can also be seen as a beat pattern on the constellations. The constellation periodically drifts from a very tight clustering to a loose clustering. For mobile 0, this beat has a period of about .5 Hz, and for mobile 1, it has a beat period of about .2 Hz. These values correspond well with the period in the respective SNR measurements. For both mobiles, the maximum penalty in SNR incurred from fractional sample offset was ~ 6 dB. At minimal fractional sample offset, the SNR was around 11 dB. This suggests that without fractional sample offset, the average SNR achievable would be on the order of 11 dB. It is important to note that the maximum possible SNR (in terms of amplitude) for the system is about 17 dB. This can been seen by looking at the difference between the signal power (~ 2 dBm) and the environmental noise power (-35 dBm). On top of this, additional noise in the system arises from the channel, amplifiers, and codecs.

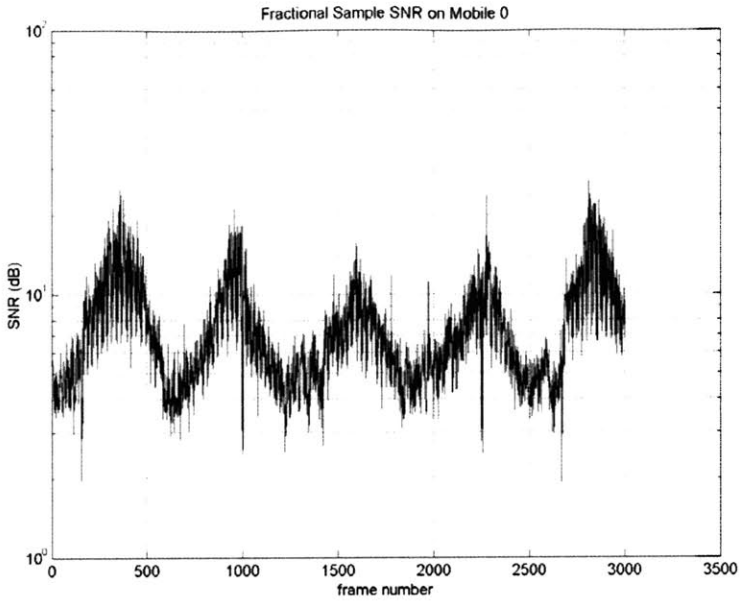


Figure 5.9 Effect of fractional sample offset on the SNR of Mobile 0 versus frame number.

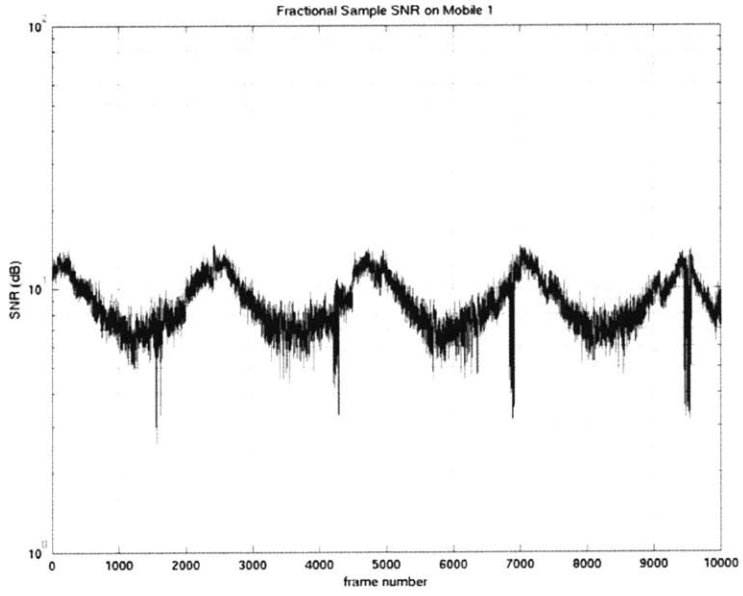


Figure 5.10 Effect of fractional sample offset on the SNR of Mobile 1 versus frame number.

There are glitches that appear quasi-periodically in the SNR graphs in figures Figure 5.9 and Figure 5.10 as well. These glitches are most likely the result of the base station clocks readjusting with the Universal Clock.

5.3 References

1. M. S. Roden, *Analog and Digital Communication Systems* (3rd ed., Prentice Hall, Englewood Cliffs, NJ 1991).

2. A. Sengupta, Bell Labs, Lucent Technologies, Murray Hill, NJ (private communication, August, 2002).

Chapter 6

Future Research and Conclusions

The successful demonstration of ADT clearly establishes it as a novel method to increase the capacity of a wireless network. The experimental results of this prototype system show that ADT can work in an indoor environment. However, analysis also indicates that there is still some additional work which could improve the performance of the system.

6.1 Future Research

There are some modifications which may improve the performance of the next-generation system. As discussed in the section 5.2 Results, fractional sample offset alone decreases SNR by 6 dB. One major improvement would thus be to remove the fractional sample offset from both the transmit and receive synchronization schemes. On the transmit side, the fractional sample synchronization could be achieved by replacing the Universal Clock with a sine wave of the same frequency. Synchronization would occur on the positive peaks of the wave, rather than rising edges. Using a sine wave would result in more exact timing information since the exact location of the maximum could be located. If the location of this maximum happened to lie between samples, the output transmission could be shifted by this fraction of a sample before copying it into the output buffer.

On the receive side, a method for compensating for fractional sample offset was derived from concepts in Orthogonal Frequency Division Multiplexing (OFDM). In OFDM implementations, the modulation of the data symbols onto orthogonal frequencies is usually done through taking an IFFT of the symbols, then modulating them onto a carrier in the time domain. This means that fractional sample timing errors result in a small frequency shift over the frame, or a linear phase component. Thus, when the symbols are transformed back to the time domain, they have simply been rotated in the complex plane. Analogously, a small time shift appears in the frequency domain as a phase rotation in the complex plane. Specifically, in the frequency domain, the ratio of the received symbols to the transmitted symbols should give the expression $e^{i\omega t_o}$, where ωt_o is the angle by which the received symbols have been rotated in phase, and t_o is the fractional sample shift. Knowing ωt_o, the received symbols can be rotated back by $-\omega t_o$, eliminating the fractional sample offset. Based on this relationship, the following method is proposed for removing the fractional sample offset. Fractional sample offset results in power leakage between symbols, leading to noise on the decoded data. In order to accurately measure the phase rotation for each frame, the timing pilot should be expanded to three adjacent symbols, with the middle slot as the actual timing pilot, and the adjacent outer slots as blanks to capture the power leakage from fractional sample offset. After decoding a frame in the normal manner, the received timing pilot symbols are used to recreate a pseudo-message with no channel pilots or data. The Fourier transform of this pseudo-frame is divided by the Fourier transform of an ideal pseudo-frame (one with an ideal timing pilot). This operation gives the angle of phase rotation. The Fourier

transform of the actual received frame can then be rotated back by this amount. It should be noted that this method increases the pilot overhead, dropping the overall throughput from 61.1 % to 50 %. However, eliminating fractional sample offset would allow the bandwidth of the slepians to be higher, potentially increasing the total possible throughput, and decreasing pilot overhead.

Another area that deserves additional investigation is the network matrix inversion algorithm. Currently, the true inverse of the matrix is calculated, and scaled, if necessary, to meet the power constraints of the system. However, this algorithm breaks down when the matrix is singular, or nearly singular. A regularized inverse could be calculated instead, which would ameliorate this problem. The regularized inverse looks at the Singular Value Decomposition (SVD) of the network matrix, and artificially adds some power to the eigenvalues which are below a certain threshold, bringing the condition of the matrix to a more favorable level. [1] However, calculating the SVD, and thus the regularized inverse, is quite computationally intensive. If possible, a less expensive algorithm for approximating the regularized inverse would be useful.

6.2 Conclusions

Adaptive Distributed Transmission was originally proposed as a novel way to increase the capacity of a wireless network, by utilizing the network of base stations as a coordinated multi-antennae. In addition to capacity gains, as a multi-antennae system, ADT also offered more robust performance for users. Initial estimates indicated that ADT should achieve good performance with about 10 dB SNR, in indoor environments. The first experimental demonstration of ADT described in this thesis has demonstrated its

ability to increase network capacity. Using readily available off-the-shelf hardware, the prototype system achieved 10^{-4} BER at about 6-7 dB SNR in terms of amplitude, which was near the theoretical performance of QPSK. Performance was decreased mainly from the noise introduced by fractional sample offset, and methods have been suggested in this dissertation, which would ameliorate this effect. In all, the experimental demonstration of ADT is an important first step to exploring the gains and challenges of this new paradigm for wireless networks.

6.3 References

1 P. Mitra, Bell Labs, Lucent Technologies, Murray Hill, NJ (private communication, August, 2002).

Appendix A – Slepian and Data Generation

```
%% generates raw data for base station to send
n=128; % samples per block
S=48; % sampling rate
B=10; % total bandwidth
gB=1; % guard bandwidth around DC
nb=1000; % number of blocks

T=n/S; % block duration
tax=[0:1/S:T-1/S]'; % time axis
eB=(B-gB)/2; % base slepian bandwidth
f0=(B+gB)/4; % amt by which to freq. shift slepians

nw=eB*T/2; % time-bandwidth product

% sleps0: original slepians                    IGNORE
% sleps1: +/- frequency-shifted copies (2x)    IGNORE
% tsleps: pulses (time-orthogonalized)         USE THIS (END PRODUCT)

[sleps0,conc]=dpss(n,nw); % slepians
sleps0=sleps0(:,find(conc>.99)); % enforce band-limitation

% now shift slepians up and down in frequency
sleps1=zeros(size(sleps0,1),1*size(sleps0,2));
for ind=1:size(sleps0,2)
  sleps1(:,2*ind-1)=sleps0(:,ind).*exp(2*pi*i*tax*f0);
  sleps1(:,2*ind)=sleps0(:,ind).*exp(-2*pi*i*tax*f0);
end

% time-orthogonalize our basis
timemat=sparse([1:n],[1:n],[0:n-1]-(n-1)/2,n,n,n)*T/n;
top=sleps1'*timemat*sleps1;

[V,D]=eig(top);

% sort by eigenvalue (time)
[Y,I]=sort(real(diag(D)));
V=V(:,I);

tsleps=sleps1*V; % END PRODUCT
tsleps0=tsleps;

% now calculate tsleps which are purely real by rotating phases
% (we can do that since their spectrum is symmetric about d.c., as
% designed - just look at tsleps(:,k) in complex plane and you'll
% see a "line" meaning we can rotate to the real axis.)

k=size(tsleps0,2); % number of degrees of freedom per block

tsleps = zeros(size(tsleps0));
for ind=1:k
  pulse = tsleps0(:,ind);
```

```
  % find angle in complex plane
  ang = mean(atan(imag(pulse)./real(pulse)));
  % rotate back to real line
  pulse1= exp(-i*ang)*pulse;
  tsleps(:,ind) = real(pulse1);
end

% constellation
const=[1+i 1-i -1-i -1+i]';
nconst=length(const);
const=const-mean(const);
const=const/sqrt(const'*const/nconst);

msgs=ceil(nconst*rand(k,2*nb)); % digital data (i.e., random)

msgs(:,1:16)=ones(18,16); % pattern to mark the beginning of data file
dat=const(msgs);

%pilots
dat(1,:)=0; % base 1 pilot
dat(2,:)=0; %  null
dat(3,:)=0; % rest of pilot matrix
dat(4,:)=0; % null
dat(5,:)=4; % slepian 10 for sync
dat(6,:)=0; % null
dat(18,:)=0; % null

%quantization of raw data
maxdat=max(max(abs(dat)));
mindat=min(min(abs(dat)));
datarange=maxdat-1;
qdat=floor((2^11-1)*(dat/datarange));


%quantization of the time slepians
maxtsleps=(max(max(abs(tsleps))));
mintsleps=(min(min(abs(tsleps))));
tsrange=maxtsleps-mintsleps;
qtsleps=floor((2^11-1)*(tsleps/tsrange));

fid=fopen('sleps.asm','w');
fprintf(fid,'\t.sect\t".slepsvals"\n');
for m=1:size(qtsleps,2)
  for ind=1:size(qtsleps,1)
    fprintf(fid,'\t.short\t%d\n',real(qtsleps(ind,m)));
    fprintf(fid,'\t.short\t%d\n',imag(qtsleps(ind,m)));
  end
end
fclose(fid);

fid=fopen('data2.asm','w');
fprintf(fid,'\t.sect\t".datablk"\n');
for m=1:size(qdat,2)
    for ind=1:size(qdat,1)
        fprintf(fid,'\t.short\t%d\n',real(qdat(ind,m)));
```

```
        fprintf(fid,'\t.short\t%d\n',imag(qdat(ind,m)));
    end
end
fclose(fid);
```

Appendix B – Base Station Code

Codec Configuration

```
<codec.h>
/*
 *  Copyright 2001 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *  U.S. Patent Nos. 5,283,900  5,392,448
 */
/* "@(#) DSP/BIOS 4.51.0 05-23-01 (barracuda-i10)" */
/*****************************************************************\
*           Copyright (C) 2000 Texas Instruments Incorporated.
*                          All Rights Reserved
*-----------------------------------------------------------------
* FILENAME..... codec.h
* DATE CREATED.. 01/05/2000
* LAST MODIFIED. 02/16/2000
* Cynthia M. Chow 07/31/2002
\*****************************************************************/

/*---------------------------------------------------------------*/
#define DSP_CTRL (*(volatile Uint32*)0x1780000)

/*---------------------------------------------------------------*/
#define CODEC_ADDR   (*(volatile Uint8*)0x01720000)
#define CODEC_DATA   (*(volatile Uint8*)0x01720004)
#define CODEC_STATUS (*(volatile Uint8*)0x01720008)
#define CODEC_PIO    (*(volatile Uint8*)0x0172000C)

/*---------------------------------------------------------------*/
extern far void CODEC_Init();
extern far void CODEC_WriteReg(Uint8 Reg, Uint8 Val, int Mce);
extern far Uint8 CODEC_ReadReg(Uint8 Reg);

/*---------------------------------------------------------------*/

/*****************************************************************\
* End of codec.h
\*****************************************************************/

<codec.c>
/*
 *  Copyright 2001 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *  U.S. Patent Nos. 5,283,900  5,392,448
 */
/* "@(#) DSP/BIOS 4.51.0 05-23-01 (barracuda-i10)" */
/*****************************************************************\
```

```
*           Copyright (C) 2000 Texas Instruments Incorporated.
*                          All Rights Reserved
*------------------------------------------------------------------------
* FILENAME..... codec.c
* DATE CREATED.. 01/05/2000
* LAST MODIFIED. 09/26/2000
* Cynthia M. Chow 07/31/2002
\************************************************************************/
#include <csl.h>
#include "codec.h"

/*-----------------------------------------------------------------------*/

/*-----------------------------------------------------------------------*/

/*-----------------------------------------------------------------------*/
void CODEC_Init() {

  volatile int temp,i,x;

  DSP_CTRL = 0x00;
  DSP_CTRL = 0x00;
  for (x=0; x<100000; x++);
  DSP_CTRL = 0x04;
  DSP_CTRL = 0x04;
  for (x=0; x<100000; x++);

  while (CODEC_ADDR & 0x80);

  /* do full calibration */
  CODEC_ADDR = 0x09 | 0x40;
  CODEC_DATA = 0xD8;
  CODEC_ADDR = 0x0B;
  while (CODEC_DATA & 0x20);

  /* set calibration mode to minimal */
  CODEC_ADDR = 0x09 | 0x40;
  CODEC_DATA = 0xC0;
  CODEC_ADDR = 0x0B;
  while (CODEC_DATA & 0x20);

  /* Modified Cynthia M. Chow
  CODEC_WriteReg( 0, 0x06, FALSE);
  //CODEC_WriteReg( 0, 0x00, FALSE);      // left AD input: unmuted,
                                          //0db gain, line input
  CODEC_WriteReg( 1, 0x06, FALSE);
  //CODEC_WriteReg( 1, 0x00, FALSE);      // right AD input:
      unmuted, 0db, line input
  CODEC_WriteReg( 2, 0x88, FALSE);  // left AUX1:muted, 0db gain
  CODEC_WriteReg( 3, 0x88, FALSE);  // right AUX1:muted, 0db gain
  CODEC_WriteReg( 4, 0x88, FALSE);  // left AUX2:muted, 0db gain
  CODEC_WriteReg( 5, 0x88, FALSE);  // right AUX2:muted, 0db gain
  CODEC_WriteReg( 6, 0x00, FALSE);  // left DAC:unmuted, 0db atten
  CODEC_WriteReg( 7, 0x00, FALSE);  // right DAC:unmuted, 0db atten
  CODEC_WriteReg( 8, 0x5C, TRUE);   // Fs and Playback:48 KHz, stereo
```

```
  //CODEC_WriteReg( 8, 0x5E, TRUE); // 32 KHz
  CODEC_WriteReg( 9, 0xC3, TRUE);   // interface config
  CODEC_WriteReg(10, 0x00, FALSE);  // int disabled, no dither, XCTL
                                    //TTL logic low
  CODEC_WriteReg(11, 0x00, FALSE);  // status reg (read only)
  CODEC_WriteReg(12, 0x40, FALSE);  // expanded features, codec ID
  CODEC_WriteReg(13, 0x00, FALSE);  // loopback disabled, 0db atten
  CODEC_WriteReg(14, 0x00, FALSE);  // playback upper reg
  CODEC_WriteReg(15, 0x00, FALSE);  // playback lower reg
  CODEC_WriteReg(16, 0x80, TRUE);   // Alt features: DAC zero, SPE
                                    // disable, SF 64-bit enh, PMCE
  CODEC_WriteReg(17, 0x00, FALSE);  // Alt features
  CODEC_WriteReg(18, 0x80, FALSE);  // LLine In:12db gain, muted mixer
  CODEC_WriteReg(19, 0x80, FALSE);  // RLine In:12db gain, mUted mixer
  CODEC_WriteReg(20, 0x00, FALSE);  //
  CODEC_WriteReg(21, 0x00, FALSE);
  CODEC_WriteReg(22, 0x00, FALSE);
  CODEC_WriteReg(23, 0x00, FALSE);
  CODEC_WriteReg(24, 0x00, FALSE);
  CODEC_WriteReg(25, 0x00, FALSE);
  CODEC_WriteReg(26, 0x00, FALSE);
  CODEC_WriteReg(27, 0x00, FALSE);
  CODEC_WriteReg(28, 0x5C, TRUE);   // Capture Data Format: stereo
  CODEC_WriteReg(29, 0x00, FALSE);
  CODEC_WriteReg(30, 0x00, FALSE);
  CODEC_WriteReg(31, 0x00, FALSE);
}

/*---------------------------------------------------------------------*/
void CODEC_WriteReg(Uint8 Reg, Uint8 Val, int Mce) {

  if  (Mce) {
    CODEC_ADDR = Reg | 0x40;
    CODEC_DATA = Val;
    while (CODEC_ADDR & 0x80);
    CODEC_ADDR = CODEC_ADDR & ~0x40;
  } else {
    CODEC_ADDR = Reg;
    CODEC_DATA = Val;
  }
  while (CODEC_ADDR & 0x80);
}

/*---------------------------------------------------------------------*/
Uint8 CODEC_ReadReg(Uint8 Reg) {

  Uint8 val;

  CODEC_ADDR = Reg;
  val = CODEC_DATA;

  return val;
}

/*---------------------------------------------------------------------*/
```

```
/*************************************************************************\
* End of codec.c
\*************************************************************************/
```

Memory Configuration

```
<hpireserve.asm>
      .sect "hpiflag"
      .int  0
      .sect "hpidata"
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0

<mylink.cmd>
-l transmitcfg.cmd

SECTIONS {
      .wfftvals:  {} > IDRAM
      .wifftvals: {} > IDRAM
}
```

DSP Code

```
<transmit.h>
//header file for transmit
#include <std.h>
// logging
#include <log.h>
#include <sts.h>
#include <trc.h>
// DSP/BIOS
#include <tsk.h>
#include <swi.h>
#include <hwi.h>
#include <idl.h>
//CSL
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include <csl_dma.h>

#include <stdlib.h>

#include "codec.h"
#include <mathf.h>
```

```
#define BLEN 128
#define LEN 256
#define DMA_IN_LEN 0x00000080
#define DMA_OUT_LEN 0x00000040
#define DRIFTDMALEN 0x0000007F
#define PAD 256
#define NCX 256 // the number of complex elements in an array
#define NX 512 // the number of elements in a complex array
#define NSLEPS 18 // the number of different slepian waveforms

// hardware initialization functions
void McBSPinit();
void DMAinitOUT();
void DMAinitIN();

//codec functions
extern void CODEC_Init();
extern void CODEC_WriteReg(Uint8 Reg, Uint8 Val, int Mce);
extern Uint8 CODEC_ReadReg(Uint8 Reg);

// miscellaneous
void cplx_mmul(int * restrict myx, short r1, short c1,
               int * restrict y,short r2, short c2,
               int *r,
               short int * restrict scale);

void avg(short int *v1, short int *v2, float w, int n);

void cplx_2x2inv(short int *k, short int *t);

int findOffset(int *vec, int len);

int updateOK(int *old_,int *new_,int eps_, int len_);

void vec_div(int *vec, int len, int fact);

// DSP/BIOS objects
extern LOG_Obj trace;
extern SWI_Obj mainSWI;
extern SWI_Obj txSWI;
extern SWI_Obj rxSWI;
extern SWI_Obj updateSWI;

/********** CSL Objects ********************************/
MCBSP_Handle mcbsp0;
MCBSP_Config config0;
DMA_Handle dma0;
DMA_Handle dma1;

//DMA functions
void configDmaGblRegs();

// SWI functions
void mainSWIfunc();
```

```
void txSWIfunc();
void rxSWIfunc();
void updateSWIfunc();

// ISRs
void dmaOutISR();
void dmaInISR();

// globals

short int *g_pout; //the transmit buffer
short int *g_p_pout; // points to the start of where the output DMA is
looking
short int *g_poutFact; // holds the array of scale factors from matrix
multiplies
short int *g_ptemp_data; // holds the array of scale factors from
matrix multiplies
short int *g_pinput1, *g_pinput2, *g_pinput_old; // input buffers to
receive synchronization clock

//short int *data;
short int *g_psleps; // points to where slepians are stored
short int *g_pmessages; // points to where the messages are stored
short int *g_pmessfrm; // holds the message to be sent

/*********************************************************/
/*matrix format [a b c d] = |a b|                        */
/*                          |c d|                        */
/*********************************************************/
short int *g_pkernel; // holds the network matrix
short int *g_pinvkernel; // holds the network matrix inverse

int *g_ppcDav; // flag to indicate that the pc has updates
short int *g_pmeasurement; // holds the update from the pc


int g_ready=0;

int g_simChan=0;  // determines if base station must simulate the
channel
int g_feedback=1; // determines if base station feedback uplink is on

void configDmaGblRegs() {
      DMA_setGlobalReg(DMA_GBL_ADDRRLDB,(Uint32)g_pout);// start by
                                                        // reloading
                                                        // out2
      DMA_setGlobalReg(DMA_GBL_ADDRRLDC,(Uint32)g_pinput1);
      DMA_setGlobalReg(DMA_GBL_CNTRLDA,DMA_OUT_LEN);
      DMA_setGlobalReg(DMA_GBL_CNTRLDB,DMA_IN_LEN);
}


/*********************************************************************/
/* McBSPinit() configures the serial port for operation with codec   */
/*********************************************************************/
```

```
void McBSPinit()
{
      mcbsp0 = MCBSP_open(MCBSP_DEV0,MCBSP_OPEN_RESET);
      if (!mcbsp0) {
            LOG_printf(&trace,"Error opening serial port.\n");
      }

      memset(config0,0,sizeof(MCBSP_Config));
      config0.spcr = 0x00002000;
      config0.rcr = 0x000100A0;
      config0.xcr = 0x000100A0;
      config0.pcr = 0x00000000;
      config0.srgr = 0x00000000;
      config0.mcr = MCBSP_MCR_DEFAULT;
      config0.rcer = MCBSP_RCER_DEFAULT;
      config0.xcer = MCBSP_XCER_DEFAULT;

      MCBSP_config(mcbsp0,&config0);

      MCBSP_enableXmt(mcbsp0);
      MCBSP_enableRcv(mcbsp0);

}

/************************************************************************/
/* DMAinitOUT() configures a DMA channel for output operation with   */
/* the serial port                                                      */
/************************************************************************/

void DMAinitOUT() {
      dma0 = DMA_open(DMA_CHA0, DMA_OPEN_RESET);
      IRQ_enable(DMA_getEventId(dma0));
      DMA_configArgs(dma0,
                  DMA_PRICTL_RMK(DMA_PRICTL_DSTRLD_NONE,
                              DMA_PRICTL_SRCRLD_B,
                              DMA_PRICTL_EMOD_NOHALT,
                              DMA_PRICTL_FS_DISABLE,
                              DMA_PRICTL_TCINT_ENABLE,
                              DMA_PRICTL_PRI_CPU,
                              DMA_PRICTL_WSYNC_XEVT0,
                              DMA_PRICTL_RSYNC_NONE,
                              DMA_PRICTL_INDEX_NA,
                              DMA_PRICTL_CNTRLD_A,
                              DMA_PRICTL_SPLIT_DISABLE,
                              DMA_PRICTL_ESIZE_32BIT,
                              DMA_PRICTL_DSTDIR_NONE,
                              DMA_PRICTL_SRCDIR_INC,
                              DMA_PRICTL_START_AUTOINIT),
                  0x00000088,
                  (Uint32)g_pout,
                  MCBSP_getXmtAddr(mcbsp0),
                  DMA_XFRCNT_RMK(DMA_XFRCNT_FRMCNT_OF(0),
                        DMA_XFRCNT_ELECNT_OF(DMA_OUT_LEN)));
```

```
    //LOG_printf(&trace,"dmaOUT started\n");
      DMA_autoStart(dma0);

}


/*********************************************************************/
/* DMAinitOUT() configures a DMA channel for output operation with   */
/* the serial port                                                   */
/*********************************************************************/

void DMAinitIN() {
      dma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);
      IRQ_enable(DMA_getEventId(dma1));
      DMA_configArgs(dma1,
                     DMA_PRICTL_RMK(DMA_PRICTL_DSTRLD_C,
                                    DMA_PRICTL_SRCRLD_NONE,
                                    DMA_PRICTL_EMOD_NOHALT,
                                    DMA_PRICTL_FS_DISABLE,
                                    DMA_PRICTL_TCINT_ENABLE,
                                    DMA_PRICTL_PRI_DMA,
                                    DMA_PRICTL_WSYNC_NONE,
                                    DMA_PRICTL_RSYNC_REVT0,
                                    DMA_PRICTL_INDEX_NA,
                                    DMA_PRICTL_CNTRLD_B,
                                    DMA_PRICTL_SPLIT_DISABLE,
                                    DMA_PRICTL_ESIZE_32BIT,
                                    DMA_PRICTL_DSTDIR_INC,
                                    DMA_PRICTL_SRCDIR_NONE,
                                    DMA_PRICTL_START_AUTOINIT),
                      0x00000188,
                      MCBSP_getRcvAddr(mcbsp0),
                      (Uint32)g_pinput1,
                      DMA_XFRCNT_RMK(DMA_XFRCNT_FRMCNT_OF(0),
                      DMA_XFRCNT_ELECNT_OF(BLEN)));

    //LOG_printf(&trace,"dmaOUT started\n");
      DMA_autoStart(dma1);

}

/*********************************************************************/
/* dmaOutISR() restarts the DMA when a block transfer to the serial  */
/* port is done                                                      */
/*********************************************************************/

void dmaOutISR() {
      static int every_other=0;
//    LOG_printf(&trace,"out: 0x%x g_p_pout: 0x%x\n",g_pout,g_p_pout);
      g_p_pout+=BLEN;
      if(g_p_pout > g_pout+3*LEN-1) {
            g_p_pout=g_pout;
      }
      DMA_setGlobalReg(DMA_GBL_ADDRRLDB,(Uint32)g_p_pout);
      if (DMA_GET_CONDITION(dma0,DMA_SECCTL_BLOCKCOND)) {
```

```
            DMA_CLEAR_CONDITION(dma0,DMA_SECCTL_BLOCKCOND);
      }
      if (DMA_GET_CONDITION(dma0,DMA_SECCTL_FRAMECOND)) {
            DMA_CLEAR_CONDITION(dma0,DMA_SECCTL_FRAMECOND);
      }
      //DMA_close(dma0);

      if (every_other==1) {
            SWI_post(&txSWI);
            every_other=0;
      }
      else {
            every_other=1;
      }

      //DMAinitOUT();

}

/*********************************************************************/
/* dmaInISR() restarts the DMA when a block transfer to the serial   */
/* port is done                                                      */
/*********************************************************************/

void dmaInISR() {
//    LOG_printf(&trace,"input: 0x%x\n",g_pinput1);
      if (DMA_GET_CONDITION(dma1,DMA_SECCTL_RDROPCOND)) {
            LOG_printf(&trace,"missing samples\n");
            exit(-1);
      }
      if (DMA_GET_CONDITION(dma1,DMA_SECCTL_BLOCKCOND)) {
            DMA_CLEAR_CONDITION(dma1,DMA_SECCTL_BLOCKCOND);
      }
      if (DMA_GET_CONDITION(dma1,DMA_SECCTL_FRAMECOND)) {
            DMA_CLEAR_CONDITION(dma1,DMA_SECCTL_FRAMECOND);
      }

      SWI_post(&rxSWI);
}

<transmit.c>
#include "transmit.h"
//#define BASE_ID 1
#define BASE_ID 0
//#define PILOT 512
//#define PIL_BITS 9
#define PILOT 1024
#define PIL_BITS 10

#define LOWHALF(x) ((x<<16)>>16) // grabs out the real half of an int
#define HIHALF(x) (x>>16) // grabs out the complex half of an int

//debug
float newScale=1;
```

```
// simulating a changing channel
short int *g_pchannels;
short int *g_ptemp;
int g_random=0;
// used to look at offset stuff in function findOffs
int g_prev=0;
int g_hisum=0;
int g_losum=0;

int g_ind=0;  // g_offs for a buffer of short ints
int g_offs=0; // the index offset of the rising clock edge in the
           // buffer (referenced as ints)
int g_oldOffs=-1; // holds the previous value of g_offs
int g_buffInd=LEN; // holds the unadjusted pointer to where the dma
                   // output transfer begins
int g_wraptx=0; // 0 is normal, 1 is txSWI should generate 2 blocks, 2
is txSWI should wait a block


short int *g_ptestreply; // holds the ideal reply (the identity matrix)
                         // from base stations

int g_start_over=0; // signals to restart from first message

int g_blkCnt=0;
// flags used in updating the network kernel
int g_newMeasurements=0; // flag to indicate that the matrix
                         // has been updated
int g_busy=0; // indicates that base stastion is using the kernel
int g_running=0; // indicates that txSWIfunc is still running
                 // from last time called

int g_flagData=0; // when 1, base stations set data in messages
                  // to 0 (indicates in data file which messages
                  // received at same time)

void main() {
      CSL_init();
      SWI_post(&mainSWI);
}

void mainSWIfunc() {

      int gie;
      int i,j;
      Uint8 v;
      short int *swap;
      g_ppcDav=(int *)0x03000000;   // flag for signaling DSP can take
meausrement
      g_pmeasurement=(short int *)0x03000004;

// the data block being sent is out1
      g_pout=(short int *)malloc(3*LEN*sizeof(short int));
      g_p_pout=g_pout;
```

```
	g_pinput1=(short int *)malloc(LEN*sizeof(short int));
	g_pinput2=(short int *)malloc(LEN*sizeof(short int));
	g_pinput_old=(short int *)malloc(LEN*sizeof(short int));
	g_poutFact=(short int *)malloc(LEN*sizeof(short int));
	g_ptemp_data=(short int *)malloc(LEN*sizeof(short int));
	g_pkernel=(short int *)malloc(8*sizeof(short int));
	g_pinvkernel=(short int *)malloc(8*sizeof(short int));
	//g_pmeasurement=(short int *)malloc(8*sizeof(short int));
	g_pmessfrm=(short int *)malloc(NSLEPS<<1*sizeof(short int));

	//debug
	g_pchannels=(short int *)malloc(8*sizeof(short int));
	g_ptemp=(short int *)malloc(NSLEPS<<1*sizeof(short int));

	g_ptestreply=(short int *)malloc(LEN*sizeof(short int));

	for (i=0,j=0;i<LEN;i++,j+=2) {
		int k;
		for (k=0;k<3;k++) {
			*g_pout=0;
			g_pout++;
		}
		g_poutFact[i]=0;
		g_ptemp_data[i]=0;
		//debug
		if (j<64) {
			g_ptestreply[j]=PILOT;
			g_ptestreply[j+1]=0;
		}
		else {
			g_ptestreply[j]=0;
			g_ptestreply[j+1]=0;
		}
	}
	g_pout-=3*LEN;
	*g_ppcDav=0;

	for (i=0;i<8;i++) {
		g_pkernel[i]=0;
		g_pmeasurement[i]=0;
	}

	// debug simulate the channel
	g_pchannels[0]=300;
	g_pchannels[1]=-111;
	g_pchannels[2]=212;
	g_pchannels[3]=111;
	g_pchannels[4]=0;
	g_pchannels[5]=0;
	g_pchannels[6]=512;
	g_pchannels[7]=0;

//	cplx_2x2inv(g_pchannels,g_pinvkernel);

	// debug commented out
```

```
      //initialize g_pkernel to the identity matrix;
      g_pkernel[0]=PILOT;
      g_pkernel[6]=PILOT;
      g_pinvkernel[0]=PILOT;
      g_pinvkernel[6]=PILOT;
      g_pmeasurement[0]=PILOT;
      g_pmeasurement[6]=PILOT;

      // map g_psleps to the place in memory where the slepians are
      g_psleps=(short int *)0x00400000;

      // map g_psleps to the place in memory where the data symbols are
      g_pmessages=(short int *)0x02000000;

      LOG_printf(&trace, "initializing hardware\n");
      gie = IRQ_globalDisable();

      CODEC_Init();
    v = CODEC_ReadReg(0x10);
    CODEC_WriteReg(0x10, v | 0x02, TRUE);

      McBSPinit();

      IRQ_globalRestore(gie);

      configDmaGblRegs();

      // start the out and in DMA channels
      DMAinitOUT();
      DMAinitIN();

//    g_p_pout+=BLEN;
//    DMA_setGlobalReg(DMA_GBL_ADDRRLDB,(Uint32)g_p_pout);

      // swap the input buffers
      swap=g_pinput1;
      g_pinput1=g_pinput_old;
      g_pinput_old=g_pinput2;
      g_pinput2=swap;
      DMA_setGlobalReg(DMA_GBL_ADDRRLDC,(Uint32)g_pinput1);


      SWI_post(&txSWI);

}

void txSWIfunc() {
      static int count=0;

      int i,j,k=0;
      int shiftInd;

      if (g_running==1) {
            LOG_printf(&trace,"txSWI still running");
            exit(-1);
```

```
	}
	g_running=1;
// invert transmission g_pkernel

	if (g_newMeasurements==1) {
		g_busy=1;
		cplx_2x2inv(g_pkernel,g_pinvkernel);
		g_busy=0;
		g_newMeasurements=0;

	}

	if (g_flagData==0) { // do the normal thing
		// mix g_pmessages
		//2*BASE_ID accesses the appropriate row in matrix inverse
		cplx_mmul((((int *)g_pinvkernel)+(2*BASE_ID)),1,2,
				(int *)g_pmessages,2,NSLEPS,
				(int *)g_pmessfrm,
					g_poutFact);
		for(i=0;i<NSLEPS<<1;i++) {
			shiftInd =abs(*g_poutFact-PIL_BITS); // 9 comes from
							// 2^9 = PILOT
			if (*g_poutFact<=PIL_BITS) {
				*g_pmessfrm= *g_pmessfrm >> shiftInd;
			}
			else {
				LOG_printf(&trace,"overflow!");
			}
			g_pmessfrm++;
			g_poutFact++;
		}
		g_poutFact-=NSLEPS<<1;
		g_pmessfrm-=NSLEPS<<1;
	}
	else {
		LOG_printf(&trace,"flagging");
		for(i=0;i<NSLEPS<<1;i++) {
			g_pmessfrm[i]=0;
			if (i==8) { // timing pilot is on 5th symbol (2*4)
				g_pmessfrm[i]=2729; // timing sync pilot
			}
		}
	}
// add pilots
	if (BASE_ID==0) {
		((int *)g_pmessfrm)[0]=PILOT;
		((int *)g_pmessfrm)[2]=0;
	}
	else {
		((int *)g_pmessfrm)[0]=0;
		((int *)g_pmessfrm)[2]=PILOT;
	}

	// test no compensating the sync pilot
	//((int *)g_pmessfrm)[4]=2729;
```

```
// simulated channel
      if (g_simChan==1) {     // if g_simChan is set, then
                              // channel is simulated

            cplx_mmul((int *)g_pchannels,1,1,
                      (int *)g_pmessfrm,1,NSLEPS,
                      (int *)g_ptemp,
                     g_poutFact);
            for(i=0;i<NSLEPS<<1;i++) {
                  shiftInd =abs(PIL_BITS-*g_poutFact); // 9 comes from
                                                       // 2^9 = PILOT
                  if (*g_poutFact<=PIL_BITS) {
                        *g_pmessfrm= *g_ptemp >> shiftInd;
                  }
                  else {
                        LOG_printf(&trace,"channel sim overflow\n");
                  }
                  g_pmessfrm++;
                  g_poutFact++;
                  g_ptemp++;
            }
            g_poutFact-=NSLEPS<<1;
            g_pmessfrm-=NSLEPS<<1;
            g_ptemp-=NSLEPS<<1;
      }

// modulates g_pmessages onto slepians
      cplx_mmul((int *)g_pmessfrm,1,NSLEPS,
                (int *)g_psleps,NSLEPS,BLEN,
                (int *)g_ptemp_data,
                g_poutFact);

      //zero out skipped samples if g_offs > g_oldOffs
      if (g_offs > g_oldOffs) {
            for(j=g_buffInd+g_oldOffs;j<g_buffInd+g_offs;j++) {
                  g_pout[j]=0;
            }
      }

      if (g_wraptx!=2) {
      // this set means that we have generated an extra block
            k=g_buffInd+g_offs;
            if (k>=3*LEN) {
                  k-=3*LEN;
            }
            if (k<0) {
                  k+=3*LEN;
            }
            //LOG_printf(&trace,"g_offs: %d k: %d",g_offs,k);

            for(j=0;j<LEN;j++) {
                  shiftInd=abs(9-*g_poutFact);
                  if (*g_poutFact<=9) {
                        *(g_pout+k)= *g_ptemp_data >> shiftInd;
```

```
            }
            else {
                  LOG_printf(&trace,"overflow\n");
            }
            g_poutFact++;
            g_ptemp_data++;
            k++;
            if (k==3*LEN) {
                  k=0;
            }
      }
      g_buffInd+=LEN;
      if (g_buffInd==3*LEN) {
            g_buffInd=0;
      }
      g_poutFact-=LEN;
      g_ptemp_data-=LEN;

      count++;
      if (count>600) {
            count=1;
      }

//DMA_setGlobalReg(DMA_GBL_ADDRRLDB,(Uint32)out1);


      //debug commented out
      if (g_start_over==1) {
            LOG_printf(&trace,"starting\n");
            g_pmessages=(short int *)0x02000000;
            for(i=0;i<8;i++) {
            // reinitialize the kernel to the identity
                  g_pkernel[i]=0;
                  g_pinvkernel[i]=0;
                  if ((i==0) || (i==6)) {
                        g_pkernel[i]=PILOT;
                        g_pinvkernel[i]=PILOT;
                  }
            }
      }
      else {
            g_pmessages+=NSLEPS<<2;
            if (g_pmessages > (short int *)0x0202327F) {
            g_pmessages = (short int *)0x02000000;
            }
      }
//LOG_printf(&trace,"out1: 0x%x\n",out1);
      if (g_wraptx==1) {
            g_running=0;
            g_wraptx=0;
            txSWIfunc();
      }
}
else {
```

```
            g_wraptx=0; // reset so that next time, will generate
waveform
            g_buffInd+=LEN;
            if (g_buffInd==3*LEN) {
                  g_buffInd=0;
            }
      }
      updateSWIfunc();

      g_running=0;
}


void rxSWIfunc() {
      //static int ignore=0;
// operates on g_pinput2
      short int *temp;
      int answer=0; // the calculated rising edge offset in the block
      static int wrap=0;

      static int firstTime=0; // determines if this is
                              // first time function has run

      int i;

      g_oldOffs=g_offs;
      g_ready=0;
      // find sync sample num
      answer=findOffset((int *)g_pinput2,BLEN);

      if ((answer==-1000) || (firstTime==0)) {
      // if findOffset found no edge or this is the first
      // time function has run
            g_offs=g_oldOffs;
            g_start_over=1;
            g_blkCnt=0;
            firstTime=1;

      }
      else {
            if (g_start_over==1) { // on the first time initialize
                                   // the offset and old offset
                  if (answer<0) {
                        g_buffInd+=LEN;
                        if (g_buffInd==3*LEN) {
                              g_buffInd=0;
                        }
                  }
                  g_offs=answer<<1;
                  g_oldOffs=g_offs;
                  g_start_over=0; // start playing through data
                  //firstTime=1;
            }
            else {
                  if ((answer<<1)-g_oldOffs>128) {
```

```
                // wrapping to the left
                    if (wrap==0) {
                        // need to generate two blocks
                        LOG_printf(&trace,"generate two blocks");
                        wrap=1;
                        g_wraptx=1;
                    }
                    g_offs=(answer-BLEN)<<1;
                }
                else {
                    if ((answer<<1)-g_oldOffs<-128) {
                    // wrapping to the right
                        if (wrap==0) {
                        // need to not generate for a block
                            LOG_printf(&trace,"skip a block");
                            wrap=1;
                            g_wraptx=2;
                        }
                        g_offs=answer<<1;
                    }
                    else {
                    // normal drift
                        if (abs((answer<<1)-g_oldOffs)!=0) {
                        // if there is a small drift, reset wrap
                            wrap=0;
                        }
                        g_offs=answer<<1;
                    }
                }
            }
        }


        //swap the input buffers for the DMA's next
        // round of auto-initialization
        temp=g_pinput1;
        g_pinput1=g_pinput_old;
        g_pinput_old=g_pinput2;
        g_pinput2=temp;

        DMA_setGlobalReg(DMA_GBL_ADDRRLDC,(Uint32)g_pinput1);

        g_ready=1;

}


void cplx_mmul(int * restrict myx, short r1, short c1,
                        int * restrict y, short r2, short c2,
                        int *r,
                        short int * restrict scale){
        short i,j,k;
        int temp1, temp2;
        int *yp;
```

```
short int sign1, sign2;
if((c1==r2) && (c1>0) && (c2>0) && (r1>0)) {
/*verify parameters*/
      yp=y;
      for(i=0; i<r1; i++) /* top to bottom */
      {
            for(j=0; j<c2; j++) /* left to right */
            {
                  yp=y+j;
                  temp1=0;
                  temp2=0;
                  sign1=0;
                  sign2=0;
                  for(k=0; k<c1; k++) /* multiply and add */
                  {

                        temp1+=_mpy(*myx,*yp)-_mpyh(*myx,*yp);
                        temp2+=_mpylh(*myx,*yp)+_mpyhl(*myx,*yp);
                        myx++;
                        yp+=c2;
                  }
                  myx-=c1;
                  sign1=16 - _norm(temp1);
                  sign2=_norm(temp2);

                  if (sign1 > 0) {
                        temp1 = (temp1 >> sign1) & 0x0000FFFF;
                        *scale = sign1;
                        scale++;
                  }
                  else {
                        temp1 = temp1 & 0x0000FFFF;
                        *scale = 0;
                        scale++;
                  }
                  if (sign2 >=16 ) {
                        temp2 = temp2 << 16;
                        *scale = 0;
                        scale++;
                  }
                  else {
                        temp2 = (temp2 << sign2) & 0xFFFF0000;
                        *scale = 16 - sign2;
                        scale++;
                  }

                  *r=_add2(temp1,temp2); /* store sum */
                  r++;
            }

            myx+=c1;
      }
}
else {
      LOG_printf(&trace,"matrix dimensions wrong\n");
```

```
	}
}

void avg(short int *v1, short int *v2, float w, int n) {
	float temp=0;
	int i;
	for(i=0;i<n;i++) {
		temp=((float)*v1)*(1-w)+((float)*v2)*w;
		*v1=_spint(temp);
		v1++;
		v2++;
	}
	v1-=n;
	v2-=n;
}

void cplx_2x2inv(short int *k, short int *t) {
/* returns -1 if the inverse would cause the system to overflow
 *     |a b|          | d -b|
 * k= |c d| inv(k)= |-c  a|*conj(det)/(det*conj(det))
*/
	float r=0, c=0;
	float r_det=0, c_det=0, det_mag=0, det_rcp=0;
	int i,j;
	float tmp[8];
	int test1=0,test2=0;
	int avg_pwr=0;
	//float ret=-1; // -1 if error, otherwise returns scale factor
	static float oldScale=-1;
	static float norm= (PILOT*PILOT)/2; // the normalization
	                                    // for the identity

	// calculate determinant
	for (i=0,j=3;i<2;i++,j--) {
		r=(float)(_mpy(*(k+2*i),*(k+2*j))-
		          _mpy(*(k+2*i+1),*(k+2*j+1)));

	c=(float)(_mpy(*(k+2*i),*(k+2*j+1))+_mpy(*(k+2*i+1),*(k+2*j)));
		tmp[2*i]=(float)*(k+2*i);
		tmp[2*i+1]=(float)*(k+2*i+1);
		tmp[2*j]=(float)*(k+2*j);
		tmp[2*j+1]=(float)*(k+2*j+1);
		if (i==0) {
			r_det+=r;
			c_det+=c;
		}
		else {
			r_det-=r;
			c_det-=c;
		}
	}

	det_mag=(r_det*r_det)+(c_det*c_det);

	// if determinant > 0, then matrix is invertable
```

```
if (det_mag > 0) {
      det_rcp=(_rcpsp(det_mag))*PILOT*PILOT;
      //det_rcp=_rsqrsp(det_mag)*PILOT;
      for(i=0;i<4;i++) {
            // this is r1+r2 because we want to multiply
            // by the complex conj of the determinant
      test1=_spint((tmp[2*i]*r_det+tmp[2*i+1]*c_det)*det_rcp);
      test2=_spint((-tmp[2*i]*c_det+tmp[2*i+1]*r_det)*det_rcp);
      /*    if ((abs(test1)>2000) || (abs(test2)>2000)) {
            // test to see if inverse is too big, overflow
                  ret=-1;
                  break;
            }*/
            //else {
                  tmp[2*i]=LOWHALF(test1);
                  tmp[2*i+1]=LOWHALF(test2);
            //}
      }

      // performing the element swapping
      for(i=0,j=3;i<2;i++,j--) {
            if (i==0) {
            // also scale by newScale
                  *(t+2*i)=_spint(tmp[2*j]);
                  *(t+2*i+1)=_spint(tmp[2*j+1]);
                  *(t+2*j)=_spint(tmp[2*i]);
                  *(t+2*j+1)=_spint(tmp[2*i+1]);
            }
            else {
                  *(t+2*i)=_spint(-tmp[2*i]);
                  *(t+2*i+1)=_spint(-tmp[2*i+1]);
                  *(t+2*j)=_spint(-tmp[2*j]);
                  *(t+2*j+1)=_spint(-tmp[2*j+1]);
            }
      }

      //find the scale factor
      for(i=0;i<4;i++) {
            avg_pwr+=(LOWHALF(*(((int *)t)+i))*
                        LOWHALF(*(((int *)t)+i)))+
                       (HIHALF(*(((int *)t)+i))*
                        HIHALF(*(((int *)t)+i)));
      }
      avg_pwr=avg_pwr>>2; // the average power in
                          // the transmission
      //LOG_printf(&trace,"power: %d",avg_pwr);
      if (oldScale==-1) {
      // the first time, just take the scale factor as is
            newScale=sqrtf((norm/((float)avg_pwr)));
      }
      else {
      // all other times, average it

newScale=.8*oldScale+.2*sqrtf((norm/((float)avg_pwr)));
      }
```

```
            oldScale=newScale;
            if (newScale<.4) {
            // this means that the new kernel has 4 times
            // the amplitude of the old
                  for(i=0;i<8;i++) {
                        *(t+i)=_spint(((float)(*(t+i)))*newScale);
                        // scale kernel inv by newScale
                  }
            }

            /*val=*((int *)t);
            *((int *)t)=*(((int *)t)+3);
            *(((int *)t)+1)=-*(((int *)t)+1);
            *(((int *)t)+2)=-*(((int *)t)+2);
            *(((int *)t)+3)=val;*/
      }
}

int maxi(int *pVec_, int len_)
/*
 * Given a vector pVec_ of atleast length len_ > 2
 * returns the index with the largest value change
 * from value right in front of it
 */
{
      int result=pVec_[1]-pVec_[0], resulti=1;
      int i;
      int temp;
      for (i=2; i<len_; ++i)
      {
            temp = pVec_[i]-pVec_[i-1];
            if (temp > result)
            {
                  result = temp; resulti = i;
            }
      }
      return resulti;
}

int maxNum(int *pVec_, int len_)
/*
 * Given a vector pVec_ of atleast length len_ > 2
 * returns the index with the largest value change
 * from value right in front of it
 */
{
      int result=pVec_[0];
      int i;
      for (i=1; i<len_; ++i)
      {
            if (pVec_[i] > result)
            {
                  result = pVec_[i];
            }
      }
```

```
      return result;
}

int minNum(int *pVec_, int len_)
/*
 * Given a vector pVec_ of atleast length len_ > 2
 * returns the index with the largest value change
 * from value right in front of it
 */
{
      int result=pVec_[0];
      int i;
      for (i=1; i<len_; ++i)
      {
            if (pVec_[i] < result)
            {
                  result = pVec_[i];
            }
      }
      return result;
}

int sumNum(int *pVec_, int len_)
/*
 * Given a vector pVec_ of atleast length len_ > 2
 * returns the index with the largest value change
 * from value right in front of it
 */
{
      int result=pVec_[0];
      int i;
      for (i=1; i<len_; ++i)
      {
            result += pVec_[i];
      }
      return result;
}

void shift(int *pVec_, int len_, int value_)
/* Moves the values in the vector pVec_ down by one
 * and inserts value_ into the last position.
 */
{
      int i;
      for(i=1; i<len_; ++i)
      {
            pVec_[i-1]=pVec_[i];
      }
      pVec_[len_-1]=value_;
}

int findOffset(int *vec, int len)
/*
 * Finds the rising edge in the buffer vec of length len.
 * Returns:
```

```
 * -1000 if no edge found
 */
{
      int i;
      int result = -1000;
      static int average[10];
      int lowSum, hiSum;
      // these are global for debug reasons
      g_prev=0; //value of sample right before rising edge sample
      g_hisum=0; //average of the 4 high sample
      g_losum=0; //average of the 4 low samples

      for(i = 0; i < len; ++i)
      {
            shift(average,10,LOWHALF(vec[i]));
            lowSum=sumNum(average, 5)-minNum(average,5)-
                  maxNum(average,5);
            hiSum=sumNum(average+5, 5)-minNum(average+5,5)-
                  maxNum(average+5,5);

            if(hiSum - lowSum> 30000) //Found some quick large
                                      // change in value
            {
                  //Find the largest change
                  result = i + (maxi(average,10)-9);
                  g_prev=result;
                  g_hisum=average[9];
                  g_losum=average[0];

            }
      }
      return result; //Found no quick large change in value
}


void updateSWIfunc() {
      int i;
      //static int blkCnt=0; // keeps track of how many updates, when
blkCnt=100, then blank out data for 50 blocks
      static int feedCnt=0;
      //debug simulating channel changing
      g_random=(*g_pmeasurement)&0x00000001;
      for (i=0;i<4;i++) {
            if (g_random == 0) {
//                g_pchannels[i]+=1;
            }
            else {
//                g_pchannels[i]-=1;
            }
      }

      //check to see if pc has data to send (g_ppcDav==1)

      if (*g_ppcDav==1) {
            if (g_busy==0) {
```

```
                if (g_feedback==1) {
                      //if (updateOK((int*)g_pkernel,
                            (int*)g_pmeasurement,5000,4)==0) {
                            //LOG_printf(&trace,"updating");
                            avg(g_pkernel,g_pmeasurement,.1,8);
                            // debug comment out
                            //avg(g_pkernel,g_pmeasurement,1,8);
                            // debug comment out
                      //}
                }

                g_newMeasurements=1;
          }
          else {
                LOG_printf(&trace,"txSWI still using
                            variable: g_pkernel\n");
                exit(-1);
          }
          *g_ppcDav=0; //lower g_ppcDav flag signals
                      // that DSP received data
          if (g_blkCnt<200) {
                g_blkCnt++;
                if (g_blkCnt==100) {
                      g_flagData=1;
                }
                if (g_blkCnt==150) {
                      g_flagData=0;
                }
          }
     }
}

int updateOK(int *old_,int *new_,int eps_, int len_) {
/* assumes old_ and new_ are same length, eps > 0
 * old_ and new_ are both natively shorts (used as ints for efficiency)
 * checks if difference between old_ and new_ real and complex
 * part is too large
 * returns -1 if check fails, 0 if update OK
 */
     int i,ret=-1;
     static int ncnt=0;
     //short int test[8];

     // tests difference between old and new elements
     for (i=0;i<len_;i++) {
          if ((abs(LOWHALF(old_[i])-LOWHALF(new_[i]))>eps_)
           || (abs(HIHALF(old_[i])-HIHALF(new_[i]))>eps_))
          {
                LOG_printf(&trace,"difference between elements
                            too great");
                break;
          }
     }
     if ((i==len_) || (ncnt==0)) {
          ret=0;
```

```
            ncnt++;
      }

      return ret;
}

void vec_div(int *vec, int len, int fact) {
// divides real and imag parts of *vec by 2^fact
// fact > 0 means divide, fact < 0 means multiply
      int i, r=0, c=0;
      for (i=0;i<len;i++) {
            if (fact>0) {
                  r=(LOWHALF(*vec))>>fact;
                  c=(((HIHALF(*vec))>>fact)<<16);
            }
            else {
                  r=(LOWHALF(*vec))<<(abs(fact));
                  c=(((HIHALF(*vec))<<(abs(fact)))<<16);
            }
            *vec=_add2(r,c);
            vec++;
      }
      vec-=len;
}
```

C++ Host PC Code

<HPIbaseStation.cpp>

```
// HPIbaseStation.cpp : Defines the entry point for the console
application.
//

/****************** change between base0 and base1 *****************/
/* change BASE_ID*/

#include "stdafx.h"
#include <stdlib.h>
#include <windows.h>
#include <evm6xdll.h>
#include <iostream>
#include <winsock2.h>
#include <time.h> // debug

//#define BASE_ID 1
#define BASE_ID 0
#define PC_SND_FLAG 0x03000000 // location of flag to DSP
#define PC_DATA 0x03000004 //  location of buffer to write data to DSP
#define LEN 16 // number of bytes to write to DSP
//#define PACKET_SIZE 12
#define PACKET_SIZE 76 // number of bytes to receive from mobiles

HANDLE h_board; // handle to DSP
LPVOID h_hpi; // handle to HPI
ULONG ul_temp;
```

```
ULONG *pcData; //buffer of data to send to DSP
ULONG dataLen; // length of pcData buffer


ULONG PC_SND_FLAG_RD() {
/* reads a single value from DSP at PC_SND_FLAG
 * returns the value read
*/
      ULONG ul_val;
      if(!evm6x_hpi_read_single(h_hpi,&ul_val,4,PC_SND_FLAG)) {
            printf("evm6x_read_single failed\n");
      }
      return ul_val;
}

void PC_SND_FLAG_SET(ULONG ul_val) {
/* writes a single value to DSP at PC_SND_FLAG
*/
      if(!evm6x_hpi_write_single(h_hpi,ul_val,4,PC_SND_FLAG)) {
            printf("evm6x_write_single failed\n");
      }
}

void PC_DATA_WRITE(ULONG *buff, ULONG *len) {
/* writes len bytes from buff to DSP at location PC_DATA
*/
      if(!evm6x_hpi_write(h_hpi,buff,len,PC_DATA)) {
            printf("evm6x_hpi_write failed\n");
      }
      else {
            if (*len!=LEN) {
                  printf("evm6x_hpi_write incomplete, wrote %d
                        bytes\n",*len);
            }
      }
}

int main(int argc, char* argv[])
{
      //debug log file to see info about network error
      FILE *fp;
      time_t t;

      // open connectino to DSP
      h_board=evm6x_open(0,FALSE);
      if(h_board==INVALID_HANDLE_VALUE) {
            printf("unable to open EVM board\n");
      }
      else {
            printf("board opened successfully\n");
      }

      // open connection to HPI
      h_hpi=evm6x_hpi_open(h_board);
```

```
if(h_hpi==NULL) {
      printf("could not open HPI port on board\n");
      evm6x_close(h_board);
}

pcData=(ULONG *)malloc(4*sizeof(ULONG));
int i;
for (i=0;i<4;i++) {
      if((i==0) || (i==3)) {
            *(pcData+i)=512;
      }
      else {
            *(((int *)pcData)+i)=0;
      }
}

WSADATA WsaDat;

if (WSAStartup(MAKEWORD(2,0),&WsaDat)!=0) {
      printf("WSA Initialization failed\n");
}

// this socket listens for incoming connections
SOCKET mySocket;
mySocket=socket(AF_INET,SOCK_STREAM,0);
if (mySocket == INVALID_SOCKET) {
      printf("Socket creation failed\n");
}

SOCKADDR_IN SockAddr;
SockAddr.sin_port = 50; // server port is port 50

SockAddr.sin_family = AF_INET;

SockAddr.sin_addr.S_un.S_un_b.s_b1=135;
SockAddr.sin_addr.S_un.S_un_b.s_b2=3;
if (BASE_ID==0) {
// base0
      SockAddr.sin_addr.S_un.S_un_b.s_b3=85;
      SockAddr.sin_addr.S_un.S_un_b.s_b4=85;
}
else {
//base1
      SockAddr.sin_addr.S_un.S_un_b.s_b3=87;
      SockAddr.sin_addr.S_un.S_un_b.s_b4=38;
}


if (bind(mySocket,
      (SOCKADDR *)&SockAddr,sizeof(SockAddr)) == SOCKET_ERROR) {
      printf("binding socket failed\n");
}

listen(mySocket,1);
```

```
fd_set readSet;
FD_ZERO(&readSet);
FD_SET(mySocket, &readSet);

fd_set cpyReadSet=readSet;

timeval delay;
delay.tv_sec =0;
delay.tv_usec = 0;

int DAV1=0, DAV2=0;     // DAV1=1, data received from mobile0,
                        // DAV2=1, data received from mobile1
ULONG *DatRecv=(ULONG *)malloc(19*sizeof(ULONG));
// each reply message has [mobile_id, pilot1, pilot2]
int RetVal=SOCKET_ERROR;
int *numRecv=(int *)malloc(2*sizeof(int));
// debug, keeps track of how many packets received from
// mobiles between DSP updates
numRecv[0]=0;
numRecv[1]=0;

int cnt=0, cnt2=0;
int selRes=0;

while(true) {

      //cpyReadSet=readSet;
      readSet=cpyReadSet;     // check all the sockets for
                              // readability
      while(selRes=select(0,&readSet,NULL,NULL,&delay) == 0) {
            if(((int)PC_SND_FLAG_RD())==0) {
            // the equality may want an int?
                  if ((DAV1==1) && (DAV2==1)) {
                  // if received replies from both mobiles
                        dataLen=LEN;
                        PC_DATA_WRITE(pcData,&dataLen);
                        PC_SND_FLAG_SET(1);
                        DAV1=0;
                        DAV2=0;
                        cnt=0;
                        cnt2=0;
                  }
            }
            readSet=cpyReadSet;
      }

      if(((int)PC_SND_FLAG_RD())==0) {
            if ((DAV1==1) && (DAV2==1)) {
            // if received replies from both mobiles
                  cnt=0;
                  cnt2=0;
                  dataLen=LEN;
                  PC_DATA_WRITE(pcData,&dataLen);
                  PC_SND_FLAG_SET(1);
                  DAV1=0;
```

```
            DAV2=0;
      }
      else {
            //printf("waiting for mobile1:%d
            mobile2:%d\n",DAV1,DAV2);
      }
}
//printf("select found something\n");

//if (select(0,&readSet,NULL,NULL,&delay)==-1) {
if (selRes==-1) {
      switch(WSAGetLastError())
      {
            case WSANOTINITIALISED:
                  printf("WSANOTINITIALIZED\n");
                  break;
            case WSAEFAULT:
                  printf("WSAEFAULT\n");
                  break;
            case WSAENETDOWN:
                  printf("WSAENETDOWN\n");
                  break;
            case WSAEINVAL:
                  printf("WSAEINVAL\n");
                  break;
            case WSAEINTR:
                  printf("WSAEINTR\n");
                  break;
            case WSAEINPROGRESS:
                  printf("WSAEINPROGRESS\n");
                  break;
            case WSAENOTSOCK:
                  printf("WSAENOTSOCK\n");
                  break;
      }
}
else {
      if (FD_ISSET(mySocket,&readSet)!=0) {
      // found a new connection
            SOCKET TempSock=SOCKET_ERROR;
            SOCKADDR_IN incoming;
            int size=sizeof(SOCKADDR_IN);
            TempSock = accept(mySocket,
                  (SOCKADDR *)(&incoming),&size);
            printf("socket port: %u\n",incoming.sin_port);
            FD_SET(TempSock, &cpyReadSet);
            // add it to the cpyReadSet so it will be
            // checked for data next time around
            printf("select found a new connection, total:
                  %d\n",(int)cpyReadSet.fd_count);
      //    readSet=cpyReadSet;
      }
      else {
      // old socket has data to read
            for (i=0;i<(int)(readSet.fd_count);i++) {
```

```
                int loop=1; // indicates data read OK
                int datRecvLen=PACKET_SIZE;
                int offs=0;
                      while(
                      ((RetVal=
                            recv(readSet.fd_array[i],(((c
                            har *)DatRecv)+offs),
                            datRecvLen,0))!=datRecvLen)
                            && (datRecvLen>0)){
                // keep trying to receive until all the
                // bytes have been read
                      if (RetVal == 0) {
                            printf("socket closed
                                  gracefully\n");

                      FD_CLR(readSet.fd_array[i],
                            &cpyReadSet);
                            closesocket(
                                  readSet.fd_array[i]);
                            loop=0; // data not read OK
                            break; // break out of while
                                  // loop
                      }
                      else {
if (RetVal==SOCKET_ERROR) {
      switch(WSAGetLastError()){
                case WSANOTINITIALISED:
                      printf("socket not initialized\n");
                      break;
                case WSAENETDOWN:
                      printf("network failure\n");
                      break;
                case WSAEFAULT:
                      printf("receive buffer not in valid
                            address space\n");
                      break;
                case WSAENOTCONN:
                      printf("socket not connected\n");
                      break;
                case WSAENETRESET:
                      printf("failure in keep alive\n");
                      break;
                case WSAENOTSOCK:
                      printf("descriptor is not a
                            socket\n");
                            break;
                case WSAESHUTDOWN:
                      printf("socket has been
                            shutdown\n");
                      break;
                case WSAEMSGSIZE:
                      printf("message truncated to fit
                            buffer\n");
                      break;
                case WSAEINVAL:
```

```
                        printf("socket not bound\n");
                        break;
                  case WSAECONNABORTED:
                        printf("connection aborted\n");
                        FD_CLR(readSet.fd_array[i],
                              &cpyReadSet);
                        closesocket(readSet.fd_array[i]);
                        break;
                  case WSAETIMEDOUT:
                        printf("connection timed out\n");
                        break;
                  case WSAECONNRESET:
                        printf("connection aborted\n");
                        FD_CLR(readSet.fd_array[i],
                        &cpyReadSet);
                        closesocket(readSet.fd_array[i]);
                        break;
                  }
                  loop=0; // data not read ok
                  break; // break out of while loop
            }
            else {
                  datRecvLen-=RetVal;
                  offs+=RetVal;
            }
      }
}

if (loop==1) {
// all 76 bytes were received
      int ind=((int)(*DatRecv));
      if ((ind>1) || (ind<0)) { // don't know why this
                              // happens, for now ignore
                        // check if bad index so program
                        // will exit gracefully and inform
            SOCKADDR_IN badSocket;
            int len=sizeof(SOCKADDR_IN);
            getsockname(readSet.fd_array[i],
                        (SOCKADDR *)(&badSocket),&len);
            fp=fopen("LOG.txt","a"); // log when things go
                                    // wrong
            time(&t);
            printf("time: %s",ctime((long *)(&t)));
            // prints the time
            fprintf(fp,"time: %s\n",ctime((long *)(&t)));
            // prints the time
            fprintf(fp,"bad index: %d from connection:
                  %u\n",ind,badSocket.sin_port);
                  // prints the port
                  for(i=0;i<19;i++) {
                        fprintf(fp,"%d ",(int)DatRecv[i]);
                        // prints all the data received
                  }
                  fclose(fp);
                  //exit(-1);
```

```
            }
      else {
            //*(pcData+2*ind)=*(DatRecv+1);
            // pilot matrix is on symbol 1 and symbol 3
            //*(pcData+(2*ind)+1)=*(DatRecv+3);
            *(pcData+ind)=*(DatRecv+1);
            *(pcData+2+ind)=*(DatRecv+3);
            *(numRecv+i)+=1;
            if (ind==0) {
                  DAV1=1; // received reply from mobile0
                  cnt++;
            }
            else {
                  DAV2=1; // received reply from mobile1
                  cnt2++;
            }
      }
}
else {
// if things are working properly, this should never occur
      printf("only received %d of %d
            bytes\n",RetVal,PACKET_SIZE);
}
/*RetVal = recv(readSet.fd_array[i],
            (char *)DatRecv,PACKET_SIZE,0);
if (RetVal == 0) {
      printf("socket closed gracefully\n");
      FD_CLR(readSet.fd_array[i],&cpyReadSet);
      closesocket(readSet.fd_array[i]);
}
else {
      if (RetVal==SOCKET_ERROR) {
            switch(WSAGetLastError())
            {
                  case WSANOTINITIALISED:
                        printf("socket not initialized\n");
                        break;
                  case WSAENETDOWN:
                        printf("network failure\n");
                        break;
                  case WSAEFAULT:
                        printf("receive buffer not in valid
                              address space\n");
                        break;
                  case WSAENOTCONN:
                        printf("socket not connected\n");
                        break;
                  case WSAENETRESET:
                        printf("failure in keep alive\n");
                        break;
                  case WSAENOTSOCK:
                        printf("descriptor is not a
                              socket\n");
                        break;
                  case WSAESHUTDOWN:
```

```
                printf("socket has been
                      shutdown\n");
                break;
          case WSAEMSGSIZE:
                printf("message truncated to fit
                      buffer\n");
                break;
          case WSAEINVAL:
                printf("socket not bound\n");
                break;
          case WSAECONNABORTED:
                printf("connection aborted\n");
                FD_CLR(readSet.fd_array[i],
                      &cpyReadSet);
                      closesocket(
                            readSet.fd_array[i]);
                      break;
          case WSAETIMEDOUT:
                printf("connection timed out\n");
                break;
          case WSAECONNRESET:
                printf("connection aborted\n");
                FD_CLR(readSet.fd_array[i],
                      &cpyReadSet);
                      closesocket(
                            readSet.fd_array[i]);
                      break;
          }
    }
    else {
          if (RetVal==PACKET_SIZE) {
          // check if received whole message
                int ind=((int)(*DatRecv));
                if ((ind>1) || (ind<0)) {
                // don't know why this happens,
                // for now ignore
                // check if bad index so program
                // will exit gracefully and inform
                SOCKADDR_IN badSocket;
                int len=sizeof(SOCKADDR_IN);
                getsockname(readSet.fd_array[i],
                (SOCKADDR *)(&badSocket),&len);
                fp=fopen("LOG.txt","a");
                // log when things go wrong
                time(&t);
                printf("time: %s",ctime(
                      (long *)(&t)));
                 // prints the time
                fprintf(fp,"time: %s\n",ctime(
                      (long *)(&t)));
                // prints the time
                fprintf(fp,"bad index: %d from
                       connection: %u\n",ind,
                      badSocket.sin_port);
                // prints the port
```

```
                                    for(i=0;i<19;i++) {
                                          fprintf(fp,"%d ",
                                                (int)DatRecv[i]);
                                          // prints all the data
                                          // received
                                    }
                                    fclose(fp);
                                    //exit(-1);
                              }
                              else {
                                    printf("msg: %d %d %d %d
                                    %d\n",((int)(*DatRecv)),
                                    (short)((*(DatRecv+1)<<16)>>16),
                                    (short)((*(DatRecv+1))>>16),
                                    (short)((*(DatRecv+3)<<16)>>16),
                                    (short)((*(DatRecv+3))>>16));
                                    *(pcData+2*ind)=*(DatRecv+1);
                        // pilot matrix is on symbol 1 and symbol 3
                                    *(pcData+(2*ind)+1)=*(DatRecv+3);
                                    *(numRecv+i)+=1;
                                    if (ind==0) {
                                          DAV1=1;
                                    // received reply from mobile0
                                          cnt++;
                                    }
                                    else {
                                          DAV2=1;
                                    // received reply from mobile1
                                          cnt2++;
                                    }
                              }
                        }
                        else {
                              printf("only received %d bytes of %d\n",
                                    RetVal,PACKET_SIZE);
                        }
                  }
            }*/
      }
}
//readSet=cpyReadSet;
}
}
return 0;
}
```

Appendix C – Mobile Code

Codec Configuration

```
<codec.h>
/*
 *  Copyright 2001 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *  U.S. Patent Nos. 5,283,900  5,392,448
 */
/* "@(#) DSP/BIOS 4.51.0 05-23-01 (barracuda-i10)" */
/*************************************************************************\
*           Copyright (C) 2000 Texas Instruments Incorporated.
*                          All Rights Reserved
*-------------------------------------------------------------------------
* FILENAME..... codec.h
* DATE CREATED.. 01/05/2000
* LAST MODIFIED. 02/16/2000
*
\*************************************************************************/

/*-------------------------------------------------------------------------*/
#define DSP_CTRL (*(volatile Uint32*)0x1780000)

/*-------------------------------------------------------------------------*/
#define CODEC_ADDR   (*(volatile Uint8*)0x01720000)
#define CODEC_DATA   (*(volatile Uint8*)0x01720004)
#define CODEC_STATUS (*(volatile Uint8*)0x01720008)
#define CODEC_PIO    (*(volatile Uint8*)0x0172000C)

/*-------------------------------------------------------------------------*/
extern far void CODEC_Init();
extern far void CODEC_WriteReg(Uint8 Reg, Uint8 Val, int Mce);
extern far Uint8 CODEC_ReadReg(Uint8 Reg);

/*-------------------------------------------------------------------------*/

/*************************************************************************\
* End of codec.h
\*************************************************************************/

<codec.c>
/*
 *  Copyright 2001 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 *  U.S. Patent Nos. 5,283,900  5,392,448
 */
/* "@(#) DSP/BIOS 4.51.0 05-23-01 (barracuda-i10)" */
/*************************************************************************\
*           Copyright (C) 2000 Texas Instruments Incorporated.
```

```
*                            All Rights Reserved
*-------------------------------------------------------------------------
* FILENAME..... codec.c
* DATE CREATED.. 01/05/2000
* LAST MODIFIED. 09/26/2000
* Cynthia M. Chow modifications 07/31/2002
\************************************************************************/
#include <csl.h>
#include "codec.h"

/*-----------------------------------------------------------------------*/

/*-----------------------------------------------------------------------*/

/*-------_-------------------------------------------------------------*/
void CODEC_Init() {

  volatile int temp,i,x;

  DSP_CTRL = 0x00;
  DSP_CTRL = 0x00;
  for (x=0; x<100000; x++);
  DSP_CTRL = 0x04;
  DSP_CTRL = 0x04;
  for (x=0; x<100000; x++);

  while (CODEC_ADDR & 0x80);

  /* do full calibration */
  CODEC_ADDR = 0x09 | 0x40;
  CODEC_DATA = 0xD8;
  CODEC_ADDR = 0x0B;
  while (CODEC_DATA & 0x20);

  /* set calibration mode to minimal */
  CODEC_ADDR = 0x09 | 0x40;
  CODEC_DATA = 0xC0;
  CODEC_ADDR = 0x0B;
  while (CODEC_DATA & 0x20);

  /* Cynthia M. Chow modifications to code */
  //CODEC_WriteReg( 0, 0x00, FALSE);
  CODEC_WriteReg( 0, 0x09, FALSE);
  //CODEC_WriteReg( 1, 0x00, FALSE);
  CODEC_WriteReg( 1, 0x09, FALSE);
  CODEC_WriteReg( 2, 0x88, FALSE);
  CODEC_WriteReg( 3, 0x88, FALSE);
  CODEC_WriteReg( 4, 0x88, FALSE);
  CODEC_WriteReg( 5, 0x88, FALSE);
  CODEC_WriteReg( 6, 0x00, FALSE);
  CODEC_WriteReg( 7, 0x00, FALSE);
  CODEC_WriteReg( 8, 0x5C, TRUE); // sampling frequency at 48kHz
  //CODEC_WriteReg( 8, 0x50, TRUE);// sampling frequency at 8KHz
  CODEC_WriteReg( 9, 0xC3, TRUE);
  CODEC_WriteReg(10, 0x00, FALSE);
```

```
  CODEC_WriteReg(11, 0x00, FALSE);
  CODEC_WriteReg(12, 0x40, FALSE);
  CODEC_WriteReg(13, 0x00, FALSE);
  CODEC_WriteReg(14, 0x00, FALSE);
  CODEC_WriteReg(15, 0x00, FALSE);
  CODEC_WriteReg(16, 0x80, TRUE);
  CODEC_WriteReg(17, 0x00, FALSE);
  CODEC_WriteReg(18, 0x80, FALSE);
  CODEC_WriteReg(19, 0x80, FALSE);
  CODEC_WriteReg(20, 0x00, FALSE);
  CODEC_WriteReg(21, 0x00, FALSE);
  CODEC_WriteReg(22, 0x00, FALSE);
  CODEC_WriteReg(23, 0x00, FALSE);
  CODEC_WriteReg(24, 0x00, FALSE);
  CODEC_WriteReg(25, 0x00, FALSE);
  CODEC_WriteReg(26, 0x00, FALSE);
  CODEC_WriteReg(27, 0x00, FALSE);
  CODEC_WriteReg(28, 0x5C, TRUE);
  CODEC_WriteReg(29, 0x00, FALSE);
  CODEC_WriteReg(30, 0x00, FALSE);
  CODEC_WriteReg(31, 0x00, FALSE);
}
/* end Cynthia M. Chow modifications */
/*-----------------------------------------------------------------------*/
void CODEC_WriteReg(Uint8 Reg, Uint8 Val, int Mce) {

  if  (Mce) {
    CODEC_ADDR = Reg | 0x40;
    CODEC_DATA = Val;
    while (CODEC_ADDR & 0x80);
    CODEC_ADDR = CODEC_ADDR & ~0x40;
  } else {
    CODEC_ADDR = Reg;
    CODEC_DATA = Val;
  }
  while (CODEC_ADDR & 0x80);
}

/*-----------------------------------------------------------------------*/
Uint8 CODEC_ReadReg(Uint8 Reg) {

  Uint8 val;

  CODEC_ADDR = Reg;
  val = CODEC_DATA;

  return val;
}

/*-----------------------------------------------------------------------*/

/************************************************************************\
* End of codec.c
\************************************************************************/
```

Memory Configuration

```
<hpisect.asm>
      .sect "hpisect1" ; place for pcdav
      .int  0
      .sect "hpisect2" ; place for
      .int  0
      .sect "hpisect3"
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      .short      0
      … (repeats for total of 720 lines)

<mylink.cmd>
-l adt2cfg.cmd

SECTIONS {
      .wfftvals:  {} > IDRAM
      .wifftvals: {} > IDRAM
}

<recData.asm>
      .sect "receivedVals"
      .short      0
      .short      0
      .short      0
      … (repeats for total of 18000 lines)
```

DSP Code

```
<r4_fft.sa>
; code from "Autoscaling Radix-4 FFT for TMS320C6000" (spra654.pdf)
; Yao-Ting Cheng, Texas Instruments Application Report, March 2000

            .title "r4_fft.sa"
            .def _r4_fft
            .text
_r4_fft .cproc n, p_x, p_w
            .reg n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
            .reg t0, t1, t2, w, x0, x1, x2, x3;
            .reg tmp, mskh, xtmph, xtmpl;
            .reg exp, scale;
            add n, 0, n2
            mvk 1, ie
            zero mskh
            mvkh 0xffff0000, mskh
            zero scale
```

```
            add n, 0, k
stage_loop:
            add n2, 0, n1
            shr n2, 2, n2
            zero ia1
            zero j
group_loop:
            add ia1, ia1, ia2
            add ia2, ia1, ia3
            add j, 0, i0
butterfly_loop:
            add i0, n2, i1
            add i1, n2, i2
            add i2, n2, i3
            ldw *+p_x[i0], x0
            ldw *+p_x[i1], x1
            ldw *+p_x[i2], x2
            ldw *+p_x[i3], x3
            add2 x1, x3, t0
            add2 x0, x2, t1
            sub2 x0, x2, t2
            add2 t0, t1, x0 ; x0
            sub2 t1, t0, t1
            ldw *+p_w[ia2], w ; load twiddle factor w2
            smpyh t1, w, tmp
            smpy t1, w, xtmph
            sub tmp, xtmph, xtmph
            and xtmph, mskh, xtmph
            smpylh t1, w, tmp
            smpyhl t1, w, xtmpl
            add tmp, xtmpl, xtmpl
            shru xtmpl, 16, xtmpl
            or xtmph, xtmpl, x2 ; x2
            sub2 x1,x3, t0
            shl t0, 16, t1
            neg t1, t1
            extu t0, 0 ,16, t0
            or t1, t0, t0
            add2 t2, t0, t1
            sub2 t2, t0, t2
            ldw *+p_w[ia1], w ; load twiddle factor w1
            smpyh t1, w, tmp
            smpy t1, w, xtmph
            sub tmp, xtmph, xtmph
            and xtmph, mskh, xtmph
            smpylh t1, w, tmp
            smpyhl t1, w, xtmpl
            add tmp, xtmpl, xtmpl
            shru xtmpl, 16, xtmpl
            or xtmph, xtmpl, x1 ; x1
            ldw *+p_w[ia3], w ; load twiddle factor w2
            smpyh t2, w, tmp
            smpy t2, w, xtmph
            sub tmp, xtmph, xtmph
            and xtmph, mskh, xtmph
```

```
            smpylh t2, w, tmp
            smpyhl t2, w, xtmpl
            add tmp, xtmpl, xtmpl
            shru xtmpl, 16, xtmpl
            or xtmph, xtmpl, x3 ; x3
            stw x0, *+p_x[i0]
            stw x1, *+p_x[i1]
            stw x2, *+p_x[i2]
            stw x3, *+p_x[i3]
            add i0, n1, i0
            cmplt i0, n, tmp
[tmp]b                  butterfly_loop ; branch to butterfly loop
            add ia1, ie, ia1
            add j, 1, j
            cmplt j, n2, tmp
[tmp]b                  group_loop ; branch to group loop
            cmpeq k, 4, tmp ; test if last stage
[tmp]b                  end ; if true, branch to end
            mvk 2, exp ; initialize exponent
            zero j ; initialize index
            mvkl 0x0000ffff, t2 ; mask for masking xtmpl
            mvkh 0x0000ffff, t2
test_bit_growth: .trip 16
            ldw *+p_x[j], tmp
            norm tmp, xtmph ; test for redundant sign bit of HI half
            shl tmp, 16, xtmpl
            norm xtmpl, xtmpl ; test for redundant sign bit of LO half
            cmplt xtmph, exp, tmp ; test if bit grow
[tmp]add xtmph, 0, exp
            cmplt xtmpl, exp, tmp ; test if bit grow
[tmp]add xtmpl, 0, exp
            cmpgt exp, 2, tmp ; if exp>2 than no scaling
[tmp]b no_scale
            cmpeq exp, 0, tmp ; compare if bit grow 3 bits
[tmp]sub 3, exp, t0 ; calculate shift
[tmp]mvk 0x0213, t1 ; csta & cstb to ext xtmpl
[tmp]add scale, t0, scale ; accumulate scale
[tmp]b scaling
            cmpeq exp, 1, tmp ; compare if bit grow 2 bit
[tmp]sub 3, exp, t0
[tmp]mvk 0x0212, t1 ; csta & cstb to ext xtmpl
[tmp]add scale, t0, scale ; accumulate scale
[tmp]b scaling
            sub 3, exp, t0 ; grows 1 bit
            mvk 0x0211, t1 ; csta & cstb to ext xtmpl
            add scale, t0, scale ; accumulate scale
            b scaling
no_scale:
            add j, 1, j
            cmplt j, n, tmp ; compare if test all output
[tmp]b      test_bit_growth ; if not, test next output
            b next_stage ; else go to next stage
scaling:
            zero j
scaling_loop: .trip 16
```

```
            ldw *+p_x[j], tmp
            shr tmp, t0, xtmph ; scaling HI half
            and xtmph, mskh, xtmph ; mask HI half
            ext tmp, t1, xtmpl ; scaling LO half
            and xtmpl, t2, xtmpl ; mask LO half by 0x0000ffff
            or xtmph, xtmpl, tmp ; x[j]=[xtmph | xtmpl]
            stw tmp, *+p_x[j]
            add j, 1, j
            cmplt j, n, tmp
[tmp]b      scaling_loop
next_stage:
            shl ie, 2, ie
            shr k, 2, k
            b stage_loop ; end of stage loop
end:
            .return scale
            .endproc

<adt2.h>

//Header file for adt2
// DSP/BIOS
#include <std.h>
#include <tsk.h>
#include <swi.h>
#include <hwi.h>
#include <idl.h>
#include <lck.h>
// logging functions
#include <log.h>
#include <sts.h>
#include <trc.h>
// CSL
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include <csl_dma.h>


#include <stdlib.h>

#include "codec.h"
//#include <mathf.h>

//Fast RTS Library
#include <fastrts67x.h>

// DSPLIB functions
#include <blk_move.h>
#include <maxidx.h>
// RTDX functions
#include <rtdx.h>
#include <target.h>

#define MOBILE_ID 0
```

```
#define DMALEN 0x00000080
#define BLEN 128 // number of samples in a block
#define LEN 256
//#define PAD 256
#define NCX 256 // the number of complex elements in an array
#define NX 512 // the number of elements in a complex array
#define NSLEPS 18 // the number of different slepian waveforms


/********** FFT functions ******************************/
extern int r4_fft(int, short*, short*);
void digitrev(int * restrict x, short int *index, int * restrict y, int
n);
void digitrev_pwr(int * restrict x, short int *index, int * restrict y,
int n);
void digitrev_index(short int *index, int n, int radix);

/********** hardware initialization functions **********/
void McBSPinit();
void DMAinitIN();
void DMAinitOUT();

void configDMAGblRegs();

extern void CODEC_Init();
extern void CODEC_WriteReg(Uint8 Reg, Uint8 Val, int Mce);
extern Uint8 CODEC_ReadReg(Uint8 Reg);


/********** software interrupt routines ****************/
void mainSWIfunc();
void syncSWIfunc();
void decodeSWIfunc();

void dataioSWIfunc();

/********** hardware  interrupt routines ***************/
void dmaOutISR();
void dmaInISR();

/********** block processing ************************/
void get_symbol(int * restrict b1, int * restrict b2, int * restrict
out, int s);
void reply(int *pilots, int *b1, int n);
void create_reply(); // NOT BEING USED


/********** math stuff ******************************/
void cplx_mmul(int * restrict myx, short r1, short c1,
                        int * restrict y,short r2, short c2,
                        int *r,
                        short int * restrict scale);

int weighAvg(float * v1, short int * v2, float a, int n);
```

```
int findMax(float *x, int j, int k);
void vec_div(int *vec, int len, int fact);
void vec_smooth(int *pVec_, int len_);
int fround(float f);
//void glitchCorrect(int *vec_, int len_);

/********** DSP/BIOS Objects ***************************/
extern LOG_Obj trace;
extern SWI_Obj mainSWI;
extern SWI_Obj syncSWI;
extern SWI_Obj dataioSWI;
extern SWI_Obj sendSWI;

/********** CSL Objects ********************************/
MCBSP_Handle mcbsp0;
MCBSP_Config config0;
DMA_Handle dma0;
DMA_Config dmaconfig0;
DMA_Handle dma1;
DMA_Config dmaconfig1;


/********** Globals ************************************/
/* arrays of complex numbers with format:              */
/* even-->real odd-->imag                              */
/*******************************************************/

// need to decide what rep for matrix


short int *g_pcurr_p; // pointer to non-bit-reversed FFT of correlation
//debug
//short int *g_psleps_p; // pointer to non-bit-reversed FFT of timing
slepian buffer

//end debug

short int *g_pinput, *g_pcurr, *g_pout1, *g_pout2, *g_pout3; // input
waveform buffers, and frame buffer
short int *g_presp, *g_presp_p, *g_pmesg; // NOT USED (output
                                          // buffers in DMA out)
short int *g_pcurr_fft, *g_psleps_fft;    // holds the FFT of timing
                                          // sync block and slepian
short int *g_presult_new; // holds the autocorrelation result
float *g_presult_old;   // holds the weighted average of
                        // autocorrelation result
short int *g_ptmp_data,*g_pmydata, *g_pfactors; // holds the scaling
                                                // factors and result
                                                // from matrix
                                                // multiplies
//int g_mydata_ind; // index of where to put data in g_pmydata
                        // (referenced in ints)

short int g_rev_index[NCX];   // length of index array is PADLEN, the
index for digit reversal
```

```
static short int *g_pzeros; // holds zeros for easy zeroing out

short int *g_pwfft; //holds the coefficients for the FFT

short int *g_psleps; // the slepian waveforms
// slepians are in PADLEN x #wvfrms format
// slepians are in 2*BUFLEN x #wvfrms format

short int *g_ptransleps; // the complex conjugated slepians

int g_count=0;     // count for how many times syncSWI has run;
int g_status=0;    // equals 1 when syncSWI is running
int g_start=0;     // the sample synchronization number

int g_dav=0;             // data available line equals 1 when a new
                         // pilot measurement has arrived

int *g_psdav; // indicates that DSP can send data to PC
int *g_pdspstart; // indicates to PC that DSP has data for it
short int *g_pdat; // holds the decoded data to send to the PC
int *g_pinfo; // sends the sample number offset of mobile

<adt2.c>

#include "adt2.h"


//extern SWI_Obj reportSWI;
//void reportSWIfunc();

//#define DRIFT_DIR 1         // mobile is faster than transmission
//#define DRIFT_DIR -1  // mobile is slower than transmission
#define END_OF_DATA 0x03057E3F
#define BEGIN_OF_DATA 0x03000000
#define DMA_IN_LEN 128
#define DMA_OUT_LEN 64
#define LOWHALF(x) ((x<<16)>>16) //grabs the lower 2 bytes (real part)
#define HIHALF(x) (x>>16) //grabs the upper 2 bytes (complex part) &
                          // makes it lower 2 bytes

//debug
//int comp=0;

//int maximum=0;
//short int *g_pmydata;
static int blknum=1;
int sampnum=0;
//debug
int off=0;
int out_start=0;
int g_driftAdj=0; // 0=do normal thing 1=decode 2 blocks 2=skip a block
int driftIgnore=0;

int *g_ptmpCorr; // holds the unscaled result of FFT(sig)*FFT(slep)
```

```
void sendToHost();


void main() {

      CSL_init();
      SWI_post(&mainSWI);

}

/*********************************************************/
/* mainSWIfunc() initializes hardware and globals        */
/*********************************************************/

void mainSWIfunc() {
      int gie, i;
      Uint8 v;
      short int *temp;

      LOG_printf(&trace, "initializing MCBSP, codec\n");
      gie = IRQ_globalDisable();

      g_psdav=(int *)0x02000000;
      g_pdspstart=(int *)0x02000004;
      g_pdat=(short int *)0x02000008;
      g_pinfo=(int *)0x020005A8;

      // initialze the buffers
      // g_pinput and decoding buffers
      g_pinput = (short int *)malloc(LEN*sizeof(short int));
      g_pcurr = (short int *)malloc(LEN*sizeof(short int));
      g_pout1 = (short int *)malloc(LEN*sizeof(short int));
      g_pout2 = (short int *)malloc(LEN*sizeof(short int));
      g_pout3 = (short int *)malloc(LEN*sizeof(short int));
      g_pcurr_p = (short int *)malloc(NX*sizeof(short int));
      g_ptmp_data = (short int *)malloc(2*NSLEPS*sizeof(short int));
//    g_pmydata = (short int *)malloc(2*NSLEPS*sizeof(short int));
      g_pfactors = (short int *)malloc(2*NSLEPS*sizeof(short int));
      g_ptmpCorr = (int *)malloc(NX*sizeof(int));

      g_pmydata=(short int *)0x03000000;
      //g_mydata_ind=0;

      // output buffers, uncomment it using codec to reply
      g_presp = (short int *)malloc(3*LEN*sizeof(short int));
      g_presp_p=g_presp;
      g_pmesg = (short int *)malloc(LEN*sizeof(short int));
      for(i=0;i<3*LEN;i++) {
            g_presp[i]=0;
      }


      //HPI sync registers and data exchange register
      *g_psdav=0;
      *g_pdspstart=0;
```

```
//      *rdy=0;
        for (i=0;i<8;i++) {
              g_pdat[i]=0;
        }

        for(i=0;i<LEN;i++) {
              g_pinput[i]=0;
              g_pcurr[i]=0;
              g_pcurr_p[2*i]=0;
              g_pcurr_p[2*i+1]=0;
              g_pout1[i]=0;
              g_pout2[i]=0;
              g_pout3[i]=0;
              g_pmesg[i]=0;
              //g_rev_index[i]=0;
        }

        g_pzeros=(short int *)malloc(LEN*sizeof(short int));
        for(i=0;i<LEN;i++) {
              g_pzeros[i]=0;
              //g_pzeros[i]=ZERO;
        }

        for(i=0;i<2*NSLEPS;i++) {
              g_ptmp_data[i]=0;
              g_pmydata[i]=0;
        }


        digitrev_index(g_rev_index, NCX, 4);

        g_pcurr_fft = (short int *) malloc(NX*sizeof(short int));
        g_psleps_fft = (short int *) malloc(NX*sizeof(short int));
        g_presult_old = (float *) malloc((NX+1)*sizeof(float));
        g_presult_new = (short int *) malloc((NX+1)*sizeof(short int));
        for(i=0;i<NX;i++) {
              g_pcurr_fft[i] = 0;
              g_psleps_fft[i] = 0;
              g_presult_old[i] = 0;
              g_presult_new[i] = 0;
        }
        g_presult_old[NX]=0;
        g_presult_new[NX]=0;

        //initialzing the w coeffs and the slepian matrix
        g_pwfft = (short int *) 0x8000E400;  //this address should be
                                             // where the wfft array is
                                             // stored in memory
        // this is where the slepians are stored. The slepians have been
        // multiplied by 2047 (2^11) to preserve precision
        g_psleps = (short int *) 0x00400000;
        // matrix is NSLEPS x BLEN (32 x 128)
        g_ptransleps = (short int *) 0x00402400;
        //g_ptransleps = (short int *) 0x00404000;
```

```
      // debug testing vec_div
      //vec_div((int *)g_psleps,BLEN,4);

      CODEC_Init();
      v = CODEC_ReadReg(0x10);
      CODEC_WriteReg(0x10, v | 0x02, TRUE);

      IRQ_globalRestore(gie);


      McBSPinit();

      configDMAGblRegs();

      dma0 = DMA_open(DMA_CHA0, DMA_OPEN_RESET);
//    dma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);// uncomment for
                                                // using codec uplink
      IRQ_enable(DMA_getEventId(dma0));
//    IRQ_enable(DMA_getEventId(dma1));   // uncomment for using codec
                                          // uplink

//    DMAinitOUT();                 // for use with codec uplink
      DMAinitIN();

      temp = g_pinput;
      g_pinput = g_pout1;
      g_pout1 = g_pcurr;
      g_pcurr = temp;

      DMA_setGlobalReg(DMA_GBL_ADDRRLDB,(Uint32)g_pinput);
}

/*********************************************************************/
/* McBSPinit() configures the serial port for operation with the     */
/* codec                                                             */
/*********************************************************************/

void McBSPinit()
{

      mcbsp0 = MCBSP_open(MCBSP_DEV0,MCBSP_OPEN_RESET);
      if (!mcbsp0) {
            LOG_printf(&trace,"Error opening serial port.\n");
      }

      memset(config0,0,sizeof(MCBSP_Config));
      config0.spcr = 0x00002000;
      config0.rcr = 0x000100A0;
      config0.xcr = 0x000100A0;
      config0.pcr = 0x00000000;
      config0.srgr = 0x00000000;
      config0.mcr = MCBSP_MCR_DEFAULT;
      config0.rcer = MCBSP_RCER_DEFAULT;
      config0.xcer = MCBSP_XCER_DEFAULT;
```

```
      MCBSP_config(mcbsp0,&config0);

      MCBSP_enableXmt(mcbsp0);
      MCBSP_enableRcv(mcbsp0);

}


void configDMAGblRegs() {
      DMA_setGlobalReg(DMA_GBL_ADDRRLDB,(Uint32)g_pinput);  // dest for
                                                            // dmaIN
      DMA_setGlobalReg(DMA_GBL_ADDRRLDC,(Uint32)g_presp);
      DMA_setGlobalReg(DMA_GBL_CNTRLDA,DMA_IN_LEN);
      DMA_setGlobalReg(DMA_GBL_CNTRLDB,DMA_OUT_LEN);
}


/*********************************************************************/
/* DMAinitOUT() configures a DMA channel for output operation with   */
/* the serial port                                                   */
/*********************************************************************/

void DMAinitOUT() {
      DMA_configArgs(dma1,
                     DMA_PRICTL_RMK(DMA_PRICTL_DSTRLD_NONE,
                                    DMA_PRICTL_SRCRLD_C,
                                    DMA_PRICTL_EMOD_NOHALT,
                                    DMA_PRICTL_FS_DISABLE,
                                    DMA_PRICTL_TCINT_ENABLE,
                                    DMA_PRICTL_PRI_DMA,
                                    DMA_PRICTL_WSYNC_XEVT0,
                                    DMA_PRICTL_RSYNC_NONE,
                                    DMA_PRICTL_INDEX_NA,
                                    DMA_PRICTL_CNTRLD_B,
                                    DMA_PRICTL_SPLIT_DISABLE,
                                    DMA_PRICTL_ESIZE_32BIT,
                                    DMA_PRICTL_DSTDIR_NONE,
                                    DMA_PRICTL_SRCDIR_INC,
                                    DMA_PRICTL_START_AUTOINIT),
      0x00000088,
      (Uint32)g_presp,
      MCBSP_getXmtAddr(mcbsp0),
      DMA_XFRCNT_RMK(DMA_XFRCNT_FRMCNT_OF(0),
                     DMA_XFRCNT_ELECNT_OF(DMA_OUT_LEN)));
      LOG_printf(&trace,"dmaOUT started\n");
      DMA_autoStart(dma1);
}

void DMAinitIN() {
      DMA_configArgs(dma0,
                     DMA_PRICTL_RMK(DMA_PRICTL_DSTRLD_B,
                     DMA_PRICTL_SRCRLD_NONE,
                     DMA_PRICTL_EMOD_NOHALT,
                     DMA_PRICTL_FS_DISABLE,
                     DMA_PRICTL_TCINT_ENABLE,
```

```
                    DMA_PRICTL_PRI_DMA,
                    DMA_PRICTL_WSYNC_NONE,
                    DMA_PRICTL_RSYNC_REVT0,
                    DMA_PRICTL_INDEX_NA,
                    DMA_PRICTL_CNTRLD_A,
                    DMA_PRICTL_SPLIT_DISABLE,
                    DMA_PRICTL_ESIZE_32BIT,
                    DMA_PRICTL_DSTDIR_INC,
                    DMA_PRICTL_SRCDIR_NONE,
                    DMA_PRICTL_START_AUTOINIT),
      //0x00000080, // block int enable
      //0x00000040, // frame int enable
      0x00000188,
      MCBSP_getRcvAddr(mcbsp0),
      (Uint32)g_pinput,
      DMA_XFRCNT_RMK(DMA_XFRCNT_FRMCNT_OF(0),
                    DMA_XFRCNT_ELECNT_OF(DMA_IN_LEN)));

      LOG_printf(&trace,"dmaIN started\n");
      DMA_autoStart(dma0);
//    }
//    else {
//          LOG_printf(&trace,"dmaIN didn't stop before trying
// to restart: %d\n",dmaCount);
            //exit(-1);
//    }
//    dmaCount++;
}

/*************************************************************************/
/* dmaOutISR() restarts the DMA when a block transfer to the serial  */
/* port is done                                                          */
/*************************************************************************/

void dmaOutISR() {
//    short int *temp;
      static int everyother=0;

      g_presp_p+=BLEN;
      if (g_presp_p == g_presp+3*LEN) {
            g_presp_p=g_presp;
      }
      DMA_setGlobalReg(DMA_GBL_ADDRRLDC,(Uint32)g_presp_p);
      if (DMA_GET_CONDITION(dma1,DMA_SECCTL_FRAMECOND)) {
            DMA_CLEAR_CONDITION(dma1,DMA_SECCTL_FRAMECOND);
      }
      if (DMA_GET_CONDITION(dma1,DMA_SECCTL_BLOCKCOND)) {
            DMA_CLEAR_CONDITION(dma1,DMA_SECCTL_BLOCKCOND);
      }
      if (everyother==0) {
      //    LOG_printf(&trace,"calling create_reply\n");
            create_reply();
            everyother=1;
      }
      else {
```

```
            everyother=0;
      }
}

/*********************************************************************/
/* dmaInISR() restarts the DMA when a block transfer from the serial */
/* port is done                                                      */
/*********************************************************************/

void dmaInISR() {
//    static unsigned int g_count = 0;

//    short int *temp;
      //Bool g_status=FALSE;
      //++g_count;
      //LOG_printf(&trace,"block received\n");
      sampnum=0;
      if (DMA_GET_CONDITION(dma0,DMA_SECCTL_RDROPCOND)) {
            LOG_printf(&trace,"still missing samples\n");
            exit(-1);
      }
      if (DMA_GET_CONDITION(dma0,DMA_SECCTL_FRAMECOND)) {
            DMA_CLEAR_CONDITION(dma0,DMA_SECCTL_FRAMECOND);
      }
      if (DMA_GET_CONDITION(dma0,DMA_SECCTL_BLOCKCOND)) {
            DMA_CLEAR_CONDITION(dma0,DMA_SECCTL_BLOCKCOND);
      }

      if (g_status!=0) {
            LOG_printf(&trace,"ERROR: syncSWI still running\n");
            DMA_stop(dma1);
            DMA_stop(dma0);
            exit(-1);
      }
      else {

            SWI_post(&syncSWI);           // debug commented out

            //LOG_printf(&trace,"finished dmaInISR\n");
      }

}



/*********************************************************************/
/* syncSWIfunc() --                                                  */
/* takes care of picking out whole symbol blocks from the incoming   */
/* sample stream                                                     */
/*********************************************************************/

void syncSWIfunc() {

      static int pk = 4; // slepian 5 is always used for
                         // synchronization
```

```
//    static float smooth=0;
      static int wrap=0; // 1 is wrapping left 2 is wrapping right
      static int wrap_echo=0; //1 is instability drift left
                              // 2 is instability drift right

      static int prev_start=0; // old start value
      static int now=0;
      short int *temp;

      //int tmp1=0,tmp2=0; // used in finding product of
                           // FFT(sig)*FFT(slep)
      int div=0; // the default division factor

      int maximum; // the index of the maximum of the autocorrelation
      static int comp; // the value of the previous index offset

      int i,k,diff;
      int scale1,scale2,scale3;
      int c,s;

//    int edge=0;

      static int print_cntl=0; // print state only once, not once per
                               // execution

      //static int g_count=0;
//    LOG_printf(&trace,"processing buffer...start: %d\n",g_start);

      g_status=1;
//    vec_div((int *)g_pcurr,BLEN,4); // to avoid FFT overflow
      vec_div((int *)g_pcurr,BLEN,1);
      vec_smooth((int *)g_pcurr,BLEN);
      if (g_count==0) {
            get_symbol((int *)g_pout1,(int *)g_pcurr,(int *)g_pout3,
                        BLEN);
      }
      else {
            get_symbol((int *)g_pout1,(int *)g_pcurr,(int *)g_pout3,
                       now);
            //get_symbol((int *)g_pout1,(int *)g_pcurr,(int *)g_pout3,
                       g_start);
      }

      // take the g_pinput buffer and zero pad so length is power of 4
      //blk_move(g_pcurr, g_pcurr_fft,LEN);
      // oldin probe point
      blk_move(g_pout3, g_pcurr_fft,LEN);
      blk_move(g_pzeros,g_pcurr_fft+LEN, LEN);

      // take the slepian and zero pad so length is power of 4
      blk_move(g_psleps+(pk*LEN), g_psleps_fft, LEN);
      blk_move(g_pzeros,g_psleps_fft+LEN, LEN);

      if (g_driftAdj==0) {
            if (wrap_echo==2) { // wrap left, when normal is to right
```

```
                if (print_cntl==0) {
                      LOG_printf(&trace,"wrap back left, opposite
                                  norm");
                      print_cntl=1;
                }
                get_symbol((int *)g_pout2,(int *)g_pout1,
                            (int *)g_pout3, g_start);
          }
          else {
                if (wrap_echo==1) {     //wrap right when normal is
                                        // to left
                      if (print_cntl==0) {
                            LOG_printf(&trace,"wrap back right
                                        opposite norm");
                            print_cntl=1;
                      }
                      get_symbol((int *)g_pout1,(int *)g_pcurr,
                                  (int *)g_pout3, g_start+BLEN);
                }
                else { // do normal
                      print_cntl=0;
                      get_symbol((int *)g_pout1,(int *)g_pcurr,
                                  (int *)g_pout3, g_start); // normal
                }
                decodeSWIfunc();
          }
    }
    else {
          if (g_driftAdj==1) { // decode two blocks from left drift
                LOG_printf(&trace,"decoding two blocks");
                get_symbol((int *)g_pout2,(int *)g_pout1,
                            (int *)g_pout3, g_start);
                decodeSWIfunc();
                get_symbol((int *)g_pout1,(int *)g_pcurr,
                            (int *)g_pout3, g_start);
                decodeSWIfunc();
                g_driftAdj=0;
          }
          else { // skip a block from right drift
                LOG_printf(&trace,"skipping a block");
                //sendToHost();
                g_driftAdj=0;
          }
    }



    scale1=r4_fft(NCX,g_pcurr_fft,g_pwfft);
    //slepsin probe point
//  digitrev((int *)g_pcurr_fft,g_rev_index,(int *)g_pcurr_p,NCX);


    scale2=r4_fft(NCX,g_psleps_fft,g_pwfft);
    //digitrev(g_psleps_fft,g_rev_index,sleps_p,NCX,4);
```

```
// doing a complex number multiply, taking the conj of the
// slepians here
// dividing by N (PADLEN) for the ifft (do it here so don't have
// to create another loop)

for(i=0;i<NCX;i++) {
// should do overflow checking here. it works now probably
// because of the >> 8
      k=i*2;
      c = ((int *)g_pcurr_fft)[i];
      s = ((int *)g_psleps_fft)[i];
      g_ptmpCorr[k]=(_mpy(c,s) + _mpyh(c,s)) >> 8;
      g_ptmpCorr[k+1]=(_mpylh(c,-s) + _mpyhl(c,s)) >> 8;
      if ((abs(g_ptmpCorr[k])>=32767) ||
            (abs(g_ptmpCorr[k+1])>=32767)) {
            div++;
      }
      //g_presult_new[k] = (_mpy(c,s) + _mpyh(c,s)) >> 8;
      //g_presult_new[k+1] = (_mpylh(c,-s) + _mpyhl(c,s)) >> 8;
}
// now scale g_ptmpCorr to shorts in g_presult_new
for(i=0;i<NX;i++) {
      g_presult_new[i] = LOWHALF((g_ptmpCorr[i])>>div);
}

// take the FFT which results in x[-n+1]
digitrev((int *)g_presult_new,g_rev_index,(int *)g_pcurr_p,NCX);

scale3=r4_fft(NCX,g_pcurr_p,g_pwfft);
// take the magnitude of the correlation squared
digitrev_pwr((int *)g_pcurr_p,g_rev_index,
             (int *)g_presult_new,NCX);

if (g_count > 1) {
      maximum=weighAvg(g_presult_old, g_presult_new, .2, NX)>>1;
      //maximum=weighAvg(g_presult_old, g_presult_new, 1, NX)>>1;
}
else {
      maximum=weighAvg(g_presult_old, g_presult_new, 1, NX)>>1;
}


comp=now;
if (maximum!=0) {
      if (maximum > BLEN) {
            if (g_count==0) {
                  now=LEN-maximum;
                  //smooth=(float)now;
            }
            else {
                  now+=LEN-maximum;
            }
      }
      else {
            if (g_count==0) {
```

```
                now=BLEN - maximum;
                //smooth=(float)now;
            }
            else {
                now-=maximum;
            }
        }
        if (now>=BLEN) {
                now-=BLEN;
                //smooth=(float)now;
        }
        else {
            if (now<0) {
                now+=BLEN;
                //smooth=(float)now;
            }
        }
        /*
        if (abs(now-comp)<BLEN) {
            smooth=.9*smooth+.1*((float)now);
            now=_spint(smooth);
        }*/
}

prev_start=g_start;


out_start=g_start+40;
if (out_start>BLEN-1) {
    out_start-=BLEN;
}

diff=comp-now; //old sync number-new sync number
if ((diff!=0) && (g_count > 0)) {
    if (diff>120) {
        if (wrap==0) {// wrapping over to the right for the
                      // first time
            g_driftAdj=2; //skip a block
            wrap=2; // right
        }
        else {
            //if ((wrap==1) && (wrap_echo==0)) {
            if (wrap==1) {
            // normally drifting left
                wrap_echo=1;
            }
            else {
            //second (or more) time drifting right
                wrap_echo=0;
            }
        }
    }
    else {
        if (diff<-120) {
            if (wrap==0) {// drifting to the left for the
```

```
                                    // first time
                              g_driftAdj=1; // decode two blocks
                              wrap=1; // left
                        }
                        else {
                              //if ((wrap==2) && (wrap_echo==0)) {
                              if (wrap==2) {
                              // normally drifting to the right
                                    wrap_echo=2;
                              }
                              else {
                                    //second (or more) time drifting
                                    // left
                                    wrap_echo=0;
                              }
                        }
                  }
                  else {
                        //if (abs(diff)>3) {
                        //    now=comp;
                        //}
                        if ((now>2) && (now<125)) { // safe to unset
                                                    // wrap
                              wrap=0;
                              wrap_echo=0; // for good measure
                        }
                  }
            }
      }
      g_start=now;

      if (g_start!=prev_start) {
            LOG_printf(&trace,"drift! %d %d ",g_start,prev_start);
      }

      if (g_start < 0) {
            LOG_printf(&trace,"ERROR: negative index\n");
            exit(-1);
      }

      temp = g_pinput;  // debug commented out
      g_pinput = g_pout2;
      g_pout2 = g_pout1;
      g_pout1 = g_pcurr;
      g_pcurr = temp;
      blk_move(g_pzeros,g_pinput,LEN); // zero out the new input buffer
      DMA_setGlobalReg(DMA_GBL_ADDRRLDB,(Uint32)g_pinput);

      //LOG_printf(&trace,"finish syncSWI\n");
//    SWI_post(&syncSWI); // debug statement
//    if (g_count<5) {
            g_count++;
//    }
      g_status=0;
      blknum++;
```

```
}

/*********************************************************************/
/* decodeSWIfunc()                                                   */
/* takes a buffer of one complete symbol and decodes with the        */
/* proper slepian                                                    */
/*********************************************************************/

void decodeSWIfunc() {

      int i,shift;
      short int *p_g_pmydata;

      //LOG_printf(&trace,"decoding symbol\n");
      // multiply block (block is in g_pout2) with transposed slepians
      cplx_mmul((int *)g_ptransleps,NSLEPS,BLEN,
                        (int *)g_pout3,BLEN,1,
                        (int *)g_ptmp_data,
                        g_pfactors);
      // shift up by factor and shift down by 2047 to get back the
      // original data
      p_g_pmydata=g_pmydata;
      for(i=0;i<NSLEPS<<1;i++) {
            shift = 11 - g_pfactors[i]; // multiplies by two as well
            //shift = 8 - g_pfactors[i]; // multiplies by 8 to
                                         // counteract divide down to
                                         //protect against FFT overflow
            if (shift >=0) {
                  *g_pmydata++= g_ptmp_data[i] >> shift;
            }
            else {
                  LOG_printf(&trace,"overflow");
            }
            if (g_pmydata==(short int *)END_OF_DATA) {
                  g_pmydata=(short int *)BEGIN_OF_DATA;
            }
      }
      g_pmydata=p_g_pmydata;

      g_dav=1;

      SWI_post(&sendSWI);

}


/*********************************************************************/
/* digitrev_index(...)                                               */
/* generates the index to bit reverse an array of complex numbers    */
/* length 256                                                        */
/* for use with the r4_fft function                                  */
/*********************************************************************/

void digitrev_index(short int *index, int n, int radix) {
      short int i;
```

```
      short int lobits, hibits, lomidbits, himidbits, result;
      lobits = 0;
      lomidbits = 0;
      himidbits = 0;
      hibits = 0;
      for(i=0;i<n;i++) {
            result = 0;
            lobits = i & 0x00000003;
            lomidbits = i & 0x000000C;
            himidbits = i & 0x0000030;
            hibits = i & 0x000000C0;
            result |= lobits << 6;
            result |= lomidbits << 2;
            result |= himidbits >> 2;
            result |= hibits >> 6;
            index[i] = result;
      }
}


/*********************************************************************/
/* digitrev(...)                                                     */
/* my function to bit reverse an array of complex numbers length 256 */
/*********************************************************************/

void digitrev(int * restrict x, short int *index, int * restrict y,
              int n) {
      int i;
      for(i=0;i<n;i++) {
            y[i] = x[index[i]];
      }
}

/*********************************************************************/
/* digitrev_pwr(...)                                                 */
/* my function to bit reverse an array of complex numbers length 256 */
/* and return the magnitude squared of the result                    */
/*********************************************************************/

void digitrev_pwr(int * restrict x, short int *index, int * restrict y,
                  int n) {
      int i,tmp;
      for(i=0;i<n;i++) {
            tmp=x[index[i]];
            y[i]=(short int)((_mpy(tmp,tmp) + _mpyh(tmp,tmp)) >> 16);
      }
}


/*********************************************************************/
/* get_symbol(...)                                                   */
/* gets the symbol from two buffers, must be in the order of         */
/* b1 = old buffer                                                   */
/* b2 = new buffer, complex vector format                            */
/*********************************************************************/
```

```
void get_symbol(int * restrict b1, int * restrict b2, int *
                  restrict out, int s) {
/* assumes b1, b2, out of length BLEN*/
      int i;
      if (s < 0) {
            LOG_printf(&trace,"negative start: %d\n",s);
            exit(-1);
      }

      b1+=s;
      for(i=s;i<BLEN;i++) {
            *out++=*b1++;
      }
      for(i=0;i<s;i++) {
            *out++=*b2++;
      }
      out=out-BLEN;
      b1=b1-BLEN;
      b2=b2-s;

}

/*********************************************************************/
/* reply(..)                                                         */
/* packs 2 pilot measurements into  block                            */
/* *r is the block                                                   */
/* val1 is pilot 1, val2 is pilot 2                                  */
/* n is the length of the block                                      */
/*********************************************************************/
// |------29 p1-----|-----29 p2-----|-----29 p3-----|-----29 p4-----|
// 0                    32                    64
//          96                    128

void reply(int *pilots, int *b1, int n) {
      int i,j;
      for (i=0,j=32;i<n>>2;i++,j++) {
            *(b1+i)=*(pilots+2*MOBILE_ID);
            *(b1+j)=*(pilots+2*MOBILE_ID+1);
      }

}

void create_reply() {
      static int p=LEN, old_start=0;
//    int off=0;
      int i;
      //check g_dav
      if (g_dav == 1) {
            reply((int *)g_pmydata,(int *)g_pmesg,BLEN);
            if (g_count<2) {
                  if (out_start<BLEN>>1) {
                        off=out_start<<1;
                  }
                  else {
```

```
                        off=(out_start-BLEN)<<1;
                  }
            }
            else {
                  off=(out_start-old_start)<<1;
                  if (off !=0) {
                        LOG_printf(&trace,"adjusting, off: %d\n",off);
                        if (off > 240) {
                              off=(out_start-BLEN)<<1;
                              //off=(BLEN-out_start)<<1;
                        }
                        if (off < -240) {
                              off+=LEN+1;
                              //off=-(out_start<<1);
                        }
                  }
            }
            old_start=out_start;
//          LOG_printf(&trace,"off: %d p: %d\n",off,p);
            p+=off;
            if (p>=3*LEN) {
                  p-=3*LEN;
                  LOG_printf(&trace,"wrapping over\n");
            }
            if (p<0) {
                  p+=3*LEN;
                  LOG_printf(&trace,"wrapping under\n");
            }
//          LOG_printf(&trace,"p: %d\n",p);
            for (i=0;i<LEN;i++) {
                  *(g_presp+p)=*g_pmesg;
                  p++;
                  g_pmesg++;
                  if(p==3*LEN) {
                        p=0;
                  }
            }
            g_pmesg-=LEN;
      }
      else {
            LOG_printf(&trace,"no pilot to measure\n");
            /*if (g_count > 1) {
                  DMA_stop(dma0);
                  DMA_stop(dma1);
                  exit(-1);
            }*/
      }

      g_dav=0;
}


/*****************************************************************/
/* cplx_mmul(...)                                                */
/* complex matrix multiply                                       */
```

```
/*****************************************************************
/

void cplx_mmul(int * restrict myx, short r1, short c1,
                        int * restrict y, short r2, short c2,
                        int *r,
                        short int * restrict scale){
      short i,j,k;
      int temp1, temp2;
      int *yp;
      short int sign1, sign2;
      if((c1==r2) && (c1>0) && (c2>0) && (r1>0)) { //verify parameters
            yp=y;
            //myx=(int *)twobytwo;
            //r=(int *)answer;
            for(i=0; i<r1; i++) /* top to bottom */
            {
                  for(j=0; j<c2; j++) /* left to right */
                  {
                        yp=y+j;
                        //temp=0;
                        temp1=0;
                        temp2=0;
                        sign1=0;
                        sign2=0;
                        for(k=0; k<c1; k++) /* multiply and add */
                        {
                              //temp+=(*x)*(*yp);
                              temp1+=_mpy(*myx,*yp)-_mpyh(*myx,*yp);
                              temp2+=_mpylh(*myx,*yp)+_mpyhl(*myx,*yp);
                              myx++;
                              yp+=c2;
                        }
                        myx-=c1;
                        sign1=16 - _norm(temp1);
                        sign2=_norm(temp2);
                        //LOG_printf(&trace,"sign1: %d\n",sign1);
                        //LOG_printf(&trace,"sign2: %d\n",sign2);
                        if (sign1 > 0) {
                              temp1 = (temp1 >> sign1) & 0x0000FFFF;
                              *scale = sign1;
                              scale++;
                        }
                        else {
                              temp1 = temp1 & 0x0000FFFF;
                              *scale = 0;
                              scale++;
                        }
                        if (sign2 >=16 ) {
                              temp2 = temp2 << 16;
                              *scale = 0;
                              scale++;
                        }
                        else {
                              temp2 = (temp2 << sign2) & 0xFFFF0000;
```

```
                        *scale = 16 - sign2;
                        scale++;
                    }

                    *r=_add2(temp1,temp2); /* store sum */
                    r++;
                }
                myx+=c1;
            }
        }
}


/************weighAvg()**************/
int weighAvg(float * v1, short int * v2, float a, int n) {
      /* v1 is where the result is put back into */
      float temp=0;
      float prev=0;
      int i;
      int res;
      for (i=0;i<n;i+=2) {
            temp= *v1*(1-a)+((float)*v2)*a;
            *v1=temp;
            // find the maximum
            if (prev < temp) {
                  prev=temp;
                  res=i;
            }
            v1+=2;
            v2+=2;
      }
      v1-=n;
      v2-=n;
      return res;
}

/************findMax()**************/
int findMax(float *x, int j, int k) {
      int i;
      int res=0;
      float temp=0;
      for (i=j;i<k;i++) {
            if (temp < *x) {
                  temp = *x;
                  res=i;
            }
            x++;
      }
      x-=k-j;
      return res;
}


void sendToHost() {
      int i;
```

```
      static int num=1; //  the number of blocks to be sent
      static int *p_g_pinfo=(int *)0x020005A8;

      *g_pdspstart=1;

//    LOG_printf(&trace,"copying decoded data: %d",num);
      for(i=0;i<NSLEPS;i++) { // copy the decoded data into the
                              // HPI exchange buffer

            *((int *)g_pdat)=*((int *)g_pmydata);
            g_pdat+=2; // these are shorts, so should increase them
                       // by 2
            g_pmydata+=2;
            if(g_pmydata>=(short int *)END_OF_DATA) {
                  g_pmydata=(short int *)BEGIN_OF_DATA;
            }
      }

      *g_pinfo=g_start;
      g_pinfo++;

      if (*g_psdav!=0) { // the PC has not dealt with the previous data
                         // in HPI buffer
            if (g_pdat>(short int *)0x020005A7) { // allow for num=20,
                                                  // if this is not
                                                  // enough, exit with
                                                  // an error
                  LOG_printf(&trace,"PC too slow: %d",num);
                  exit(-1);
            }
            else {
                  num++;
            }
      }
      else {
            // PC has read the last data, so restart HPI buffer from
            // the beginning
            // restart the num count
            *g_psdav=num*(NSLEPS<<2); // number of blocks*number of
                                      // symbols*bytes/symbol
            g_pdat=(short int *)0x02000008;
            g_pinfo=p_g_pinfo; // restart the offset log
            //if (*g_psdav>72) {
            //    LOG_printf(&trace,"sending multiple frames: %d",num);
            //}
            num=1;
      }
}

int fround(float f) {
      int res=0;
//    int tmp=0;
//    tmp=(int)(f*2);
      res=((int)(f*2))-(int)f;
      return res;
```

```
}

void vec_div(int *vec, int len, int fact) {
// divides the vector vec of length len by 2^fact, fact > 0
      int i, r=0, c=0;
      for (i=0;i<len;i++) {
            //r=(((*vec)&0x0000FFFF)<<16)>>(16+fact);
            //c=(((*vec)>>(16+fact))<<16);
            r=LOWHALF(*vec)>>fact;
            c=(HIHALF(*vec)>>fact)<<16;
            *vec=_add2(r,c);
            vec++;
      }
      vec-=len;
}

void vec_smooth(int *pVec_, int len_)
//Assumes that len_ is atleast 2.
{
      int i, temp1, temp2;
      for (i=0;i<len_;i++)
      {
            if((abs(HIHALF(pVec_[i]))>1000) ||
              (abs(LOWHALF(pVec_[i])) >1000))
            {
                  LOG_printf(&trace,"smoothing glitch");
                  temp1=pVec_[(i+1) >= len_ ? i-1:i+1];
                  temp2=pVec_[(i-1) < 0 ? i+1:i-1];
                  pVec_[i]=_add2(temp1,temp2);

      pVec_[i]=_add2(LOWHALF(pVec_[i])>>1,((HIHALF(pVec_[i]))>>1)<<16);
            }
      }
}
```

C++ Host PC Code

<HPIreceive.cpp>

```
// HPIreceive.cpp : Defines the entry point for the console
application.
//
/****** this needs to be changed between mobile 1 and 0 *************/
/* change MOBILE_ID*/

#include "stdafx.h"
#include <stdlib.h>
#include <windows.h>
#include <evm6xdll.h>
#include <iostream>
#include <winsock2.h>


#define DSP_SND_FLAG 0x02000000 // address of data ready flag
#define DSP_START_FLAG 0x02000004 // address of DSP started flag
#define DSP_DATA 0x02000008 // address of DSP_DATA buffer
```

```
#define DSP_OFFSET 0x020005A8
#define LEN 72    // length in bytes of a data frame from DSP
#define PACKET_SIZE 76 // length in bytes of one data frame + MOBILE ID
//#define MOBILE_ID 0 // mobile ID number
 #define MOBILE_ID 1 // mobile ID number

HANDLE h_board; // handle to the DSP board
LPVOID h_hpi; // handle to the HPI interface
ULONG ul_temp;

ULONG *respPacket; // buffer holding the data to be sent to the base
stations and display
ULONG packetLen; // length in bytes of the received response plus
mobile ID

ULONG *dspResp; // buffer holding the contents of data read from DSP
ULONG respLen=0; // length in bytes of response received from the DSP


ULONG DSP_FLAG_RD(ULONG addr) {
/* reads a single int from the location addr */
      ULONG ul_val;
      if(!evm6x_hpi_read_single(h_hpi,&ul_val,4,addr)) {
            printf("evm6x_read_single() failed\n");
      }
      return ul_val;
}

void DSP_FLAG_SET(ULONG addr,ULONG ul_val) {
/* writes the value ul_val to addr */
      if(!evm6x_hpi_write_single(h_hpi,ul_val,4,addr)) {
            printf("evm6x_write_single() failed\n");
      }
}

void DSP_DATA_RD(ULONG *buff, ULONG *len) {
/* reads len bytes from DSP_DATA to buff */
      ULONG temp=*len;
      if(!evm6x_hpi_read(h_hpi,buff,len,DSP_DATA)) {
            printf("evm6x_hpi_read() failed\n");
      }
      else {
            if (*len != temp) {
                  printf("evm6x_read incomplete, read %d bytes of
                        %d\n",*len,temp);
            }
      }
//    printf("%d %d %d %d\n",(int)((*buff)<<16)>>16,(int)(*buff)>>16,
            (int)((*(buff+1))<<16)<<16,(int)(*(buff+1))>>16);
}



int main(int argc, char* argv[])
{
```

```
int i,j;
// holds data from DSP, max size is 20 frames
dspResp=(ULONG *)malloc(20*18*sizeof(ULONG));
//holds data to send, max size is 60 response packets;
respPacket=(ULONG *)malloc(80*19*sizeof(ULONG));
// the index at where to start writing the data to respPacket
int respInd=0;
// holds the address of respPacket
ULONG *p_respPacket=respPacket;
ULONG *p_dspResp=dspResp; // holds the address of dspResp
for (i=0;i<80;i++) {
      *respPacket=MOBILE_ID; // create a default reponse packet
      respPacket++;
      for(j=0;j<18;j++) {
            *respPacket=0;
            respPacket++;
      }
}
respPacket=p_respPacket;

// HPI initialization
h_board=evm6x_open(0,FALSE);
if(h_board==INVALID_HANDLE_VALUE) {
      printf("unable to open Evm board\n");
}
else {
      printf("board opened successfully\n");
}

h_hpi=evm6x_hpi_open(h_board);
if(h_hpi==NULL) {
      printf("could not open HPI port on board\n");
      evm6x_close(h_board);
}


// open a socket connection to base stations
WSADATA WsaData;
if (WSAStartup(MAKEWORD(2,0),&WsaData)!=0) {
      printf("WSA Initialization failed\n");
}

// base 0
SOCKET mySocket;
mySocket = socket(AF_INET,SOCK_STREAM,0);
if (mySocket == INVALID_SOCKET) {
      printf("could not create socket to base 0\n");
}
SOCKADDR_IN SockAddr;
SockAddr.sin_port=50;
SockAddr.sin_family=AF_INET;

SockAddr.sin_addr.S_un.S_un_b.s_b1=135;
SockAddr.sin_addr.S_un.S_un_b.s_b2=3;
SockAddr.sin_addr.S_un.S_un_b.s_b3=85;
```

```
SockAddr.sin_addr.S_un.S_un_b.s_b4=85;

// base 1
SOCKET mySocket2;
mySocket2 = socket(AF_INET,SOCK_STREAM,0);
if (mySocket2 == INVALID_SOCKET) {
      printf("could not create socket to base 1\n");
}
SOCKADDR_IN SockAddr2;
SockAddr2.sin_port=50;
SockAddr2.sin_family=AF_INET;

SockAddr2.sin_addr.S_un.S_un_b.s_b1=135;
SockAddr2.sin_addr.S_un.S_un_b.s_b2=3;
SockAddr2.sin_addr.S_un.S_un_b.s_b3=87;
SockAddr2.sin_addr.S_un.S_un_b.s_b4=38;

// display server
SOCKET mySocket3;
mySocket3 = socket(AF_INET,SOCK_STREAM,0);
if (mySocket3 == INVALID_SOCKET) {
      printf("could not create socket to display\n");
}
SOCKADDR_IN SockAddr3;
if (MOBILE_ID==0) {
// mobile0
      SockAddr3.sin_port=51;
}
else {
//mobile1
      SockAddr3.sin_port=50;
}
SockAddr3.sin_family=AF_INET;

SockAddr3.sin_addr.S_un.S_un_b.s_b1=135;
SockAddr3.sin_addr.S_un.S_un_b.s_b2=3;

if (MOBILE_ID==1) {
      SockAddr3.sin_addr.S_un.S_un_b.s_b3=85;
      SockAddr3.sin_addr.S_un.S_un_b.s_b4=95;
}
else {
      SockAddr3.sin_addr.S_un.S_un_b.s_b3=83;
      SockAddr3.sin_addr.S_un.S_un_b.s_b4=209;
}

if (connect(mySocket,
            (SOCKADDR *)&SockAddr,sizeof(SockAddr)) != 0) {
      printf("connecting to base0 failed, error:
            %d\n",WSAGetLastError());
}
if (connect(mySocket2,
            (SOCKADDR *)&SockAddr2,sizeof(SockAddr2)) != 0) {
      printf("connecting to base1 failed, error:
            %d\n",WSAGetLastError());
```

```
        }
        if (connect(mySocket3,
                    (SOCKADDR *)&SockAddr3,sizeof(SockAddr3)) != 0) {
              printf("connecting to display failed, error:
                    %d\n",WSAGetLastError());
        }
//      else {
              fd_set writeSet;
              fd_set oldwriteSet;
              FD_ZERO(&writeSet);
              FD_SET(mySocket,&writeSet);
              FD_SET(mySocket2,&writeSet);
              FD_SET(mySocket3,&writeSet);
              oldwriteSet=writeSet;
              timeval delay;
              delay.tv_sec=0;
              delay.tv_usec=0;

              // disable Nagle's algorithm so won't buffer small sends
              bool type=true;
              int len=4;
              setsockopt(mySocket,IPPROTO_TCP,TCP_NODELAY,
                          (const char *)&type,len);
              getsockopt(mySocket,IPPROTO_TCP,TCP_NODELAY,
                          (char *)&type,&len);
              if(type) {
                    printf("Nagle algorithm disabled\n");
              }
              else {
                    printf("Nagle algorithm enabled\n");
              }

              type=true;
              setsockopt(mySocket2,IPPROTO_TCP,TCP_NODELAY,
                          (const char *)&type,len);
              getsockopt(mySocket2,IPPROTO_TCP,TCP_NODELAY,
                          (char *)&type,&len);
              if(type) {
                    printf("Nagle algorithm disabled\n");
              }
              else {
                    printf("Nagle algorithm enabled\n");
              }

/*            type=true;
              setsockopt(mySocket3,IPPROTO_TCP,TCP_NODELAY,
                    (const char *)&type,len);
              getsockopt(mySocket3,IPPROTO_TCP,TCP_NODELAY,
                    (char *)&type,&len);
              if(type) {
                    printf("Nagle algorithm disabled\n");
              }
              else {
                    printf("Nagle algorithm enabled\n");
              }*/
```

```
while(DSP_FLAG_RD((ULONG)DSP_START_FLAG)==0){}
printf("DSP started\n");

int newData=0;

while(true) {
      respLen=DSP_FLAG_RD((ULONG)DSP_SND_FLAG);
      if((((int)respLen)!=0) && (respInd<=1501)){
            packetLen+=respLen;
            DSP_DATA_RD(dspResp,&respLen);
            if (respLen!=0) {
                  printf("buffering %d\n",packetLen);
            }
            for (j=0;j<(((int)respLen)/72);j++) {
            //  for number of blocks from DSP
                  *(respPacket+respInd)=(ULONG)MOBILE_ID;
                  // set first element to be MOBILE_ID;
                  //respPacket++;
                  respInd++;
                  packetLen+=4;
                  for(i=0;i<18;i++) {
                  // set the rest of block to data
                        *(respPacket+respInd)=*dspResp++;
                        respInd++;
                  }
            }
            //respPacket=p_respPacket;
            dspResp=p_dspResp;
            newData=3;
             // let sockets know have new data to send
            DSP_FLAG_SET((ULONG)DSP_SND_FLAG,0);
            //  let DSP know received data OK

      }
      else {
      //    if ((int)respLen>0) {
      //          printf("respLen: %d respInd: %d
                              packetLen:d\n",(int)respLen,
                              respInd,(int)packetLen);
      //    }
      }
      // initialize the writeSet to check all the sockets
      // each time
      writeSet=oldwriteSet;

      while(select(0,NULL,&writeSet,NULL,&delay)==0) {
            respLen=DSP_FLAG_RD((ULONG)DSP_SND_FLAG);
            if((((int)respLen)!=0) && (respInd<=1501)){
            //if((((int)respLen)!=0) && (newData==0)) {
            // don't do it again if did it the first time
                  packetLen+=respLen;
                  DSP_DATA_RD(dspResp,&respLen);
                  if (respLen!=0) {
                        printf("buffering %d\n",packetLen);
```

```
                        }

                        for (j=0;j<(((int)respLen)/72);j++) {
                        //  for number of blocks from DSP
                        *(respPacket+respInd)=(ULONG)MOBILE_ID;
                        // set first element to be MOBILE_ID;
                              respInd++;
                              //respPacket++;
                              packetLen+=4;
                  ←for(i=0;i<18;i++) { // set the rest of block to data
                        *(respPacket+respInd)=*dspResp++;
                        respInd++;
                  }
            }
            //respPacket=p_respPacket;
            dspResp=p_dspResp;
            newData=3; // let sockets know have new data to send
            //  let DSP know received data OK
            DSP_FLAG_SET((ULONG)DSP_SND_FLAG,0);
      }
      else {
      //    if ((int)respInd>0) {
      //          printf("respLen: %d respInd: %d packetLen:
                        %d\n",(int)respLen,respInd,(int)packetLen);
            //    }
      }
      //printf("waiting for socket to be free\n");
      writeSet=oldwriteSet;
}

int RetVal=0;

if (newData!=0)  {
// update needs to be sent, or display data needs to be sent, or both
      int len=0;
      for (i=0;i<(int)(writeSet.fd_count);i++) {
            //RetVal=SOCKET_ERROR;
            if ((writeSet.fd_array[i]==mySocket) ||
                  (writeSet.fd_array[i]==mySocket2)) {
            // send only one frame to the base stations
                  if (newData<=2) {
                        len=PACKET_SIZE;
                        while(len>0) {
                              RetVal = send(writeSet.fd_array[i],
                                    (char *)respPacket,len,0);
                              if (RetVal == SOCKET_ERROR) {
                                    printf("BASE error code:
                                    %d\n",WSAGetLastError());
                                    break;
                              }
                              else {
                                    len-=RetVal;
                              }
                              printf("sending to base %d
                                    bytes\n",PACKET_SIZE);
```

```
                        }
                        //if (RetVal!=PACKET_SIZE) {
                        //    printf("sent %d bytes to base
                                    station\n",RetVal);
                        //}
                        //printf("sent to base\n");
                        // means the update has already occured
                        newData--;
                  }
            }
            else {
            // send all the buffered frames to the display
            //printf("send to display %d
                   bytes\n",(int)packetLen);
            len=(int)packetLen;
            while(len>0) {
                  RetVal = send(writeSet.fd_array[i],
                  (char *)respPacket,len,0);
                  if (RetVal == SOCKET_ERROR) {
                        printf("DISPLAY error code:
                              %d\n",WSAGetLastError());
                        break;
                  }
                  //if (RetVal!=(int)packetLen) {
                  //    printf("sent %d bytes to
                              display\n",RetVal);
                  //}
                  else {
                        len-=RetVal;
                  }
            }
            printf("sent to display  %d bytes\n",(int)packetLen);
            packetLen=0;
            respInd=0;
            newData=0; // there is nothing new to try and send
      }
   }
   //newData=0;
}
}
return 0;
}
```