SOFTWARE SIMULATION OF THE INFOPAD WIRELESS DOWNLINK

by

Craig M. Teuscher

Memorandum No. UCB/ERL M95/16

6 March 1995

SOFTWARE SIMULATION OF THE INFOPAD WIRELESS DOWNLINK

by

Craig M. Teuscher

Memorandum No. UCB/ERL M95/16

6 March 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SOFTWARE SIMULATION OF THE INFOPAD WIRELESS DOWNLINK

by

Craig M. Teuscher

Memorandum No. UCB/ERL M95/16

6 March 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Acknowledgements

I would like to express thanks to Professor Robert Brodersen for his advice and support which have made this research effort possible. I am also grateful to Sam Sheng for numerous technical discussions, research direction, and his invaluable computer expertise, especially with software tools employed in this research.

Special thanks are due my grandparents, Chad and Jennie White, for making the first two and one half years of graduate school such a memorable experience. Thank you for the warmth, kindness and atmosphere you have provided. To my parents, Lynn and Cheryl Teuscher, I am deeply indebted for the twenty-seven years of training, guidance, commitment and love you have invested in me. To a great extent, my life is the product of your sacrifice and example. That I might prove to be a favorable reflection is my ongoing hope.

Finally, I would like to acknowledge the faith, support, and patience of my wife Gayle. Thank you for providing the relationship which so enriches and animates my life. Your quiet confidence, trust, and love have substantially assisted in this research process. For that I am most grateful.

Table of Contents

1 Introduction

Spread spectrum techniques have received much recent attention for use in commercial code-division multiple access (CDMA) communication systems. Of particular note is the present QUALCOMM proposal to implement a direct sequence spread-spectrum (DS-SS) digital cellular telephony system with substantial capacity increases over existing analog systems [Gil91, Qual92]. Another obvious extension of the spread-spectrum approach is application to an indoor wireless channel. In ongoing research at UC Berkeley, students involved in the InfoPad project are investigating the feasibility of such a system [Sheng91]. The project's goal is to design and construct a prototype system capable of delivering up to 2 Megabits per second of data per user from ceiling-mounted base stations to as many as fifty simultaneous users per basestation. Each user employs a low-power portable terminal equipped with an RF transceiver; system design assumes an office space environment. The base stations are assumed to be interconnected by a high speed fiber-optic backbone network. While traditional spread-spectrum communication system design is well-understood, several feasibility issues must be reexamined at these bit rates in light of present technological, power, and cost constraints.

This research uses system level software simulation to model the InfoPad downlink in order to address three basic issues: mobile receiver synchronization, multipath equalization, and bit error rate performance on the link. Relevant background material and references for this research are provided along with a detailed description of the author's contributions. This paper will not review the basic theory of spread spectrum communications, but excellent tutorials may be found in [Harris73, Pick82]. A brief overview of the InfoPad system proposal is provided in the second chapter. Chapter 3 examines chip-level synchronization issues, while chapter 4 is devoted to interference and diversity combining. Simulation model development and results are presented in the fifth chapter. Lastly, conclusions are drawn and future work is proposed in chapter 6.

This paper will show that simple hardware implementations should be sufficient to enable mobile receiver synchronization and tracking for the InfoPad. In addition, the analysis demonstrates that the traditional RAKE receiver structure must be modified in an interference-limited environment, such as that used in InfoPad; maximal ratio combining under these circumstances yields minimal bit error rate (BER) gains. Finally, this research provides a simulation infrastructure permitting investigation of other system design issues.

2 System Overview

The InfoPad system design employs an asymmetrical radio link between mobile users and the wired network. In the present conception, the mobile user accesses high-bandwidth information services via the wireless link between ceiling-mounted base stations and his own untethered, battery operated terminal. Implicit is the assumption that the information flowing to the mobile user is of significantly higher bandwidth than the information flowing from the user. On the downlink (from base-station to mobile) the InfoPad project aims to support a data rate of up to 2 Megabits per second per user, enabling the transmission of compressed full-motion color video, graphics, audio, text, and data. The uplink (mobile to base station) is assumed to carry much lower bandwidth information, consisting primarily of terminal inputs such as pen strokes and voice commands. This relaxes the bit rate requirements on the uplink, requiring less than sixty-four kilobits per second per user.

With the current availability of inexpensive, integrated commercial radios which satisfy the uplink bit rate requirements, this problem has largely been solved. The more challenging design concerns the downlink implementation where bit rate requirements are significantly more demanding. Results presented in this paper focus exclusively on the downlink and is limited in scope to the investigation of a single cell. Although the complete system proposal calls for a picocellular approach with multiple base stations, most issues relating to multiple base station interaction are beyond the scope of this paper. Discussions or derivations in which base station multiplicity is relevant will be noted. However, the primary focus of the ensuing discussion will be a single, isolated downlink cell. Following is a brief description of the overall link design.

2.1 System Description

2. 1. 1 Multiplexing Users via Walsh Functions

The proposed system design for the InfoPad project is a scaled version of the U.S. IS-95 digital cellular CDMA standard. Simplified transmitter and receiver block diagrams are shown below in Figure 2.1 and Figure 2.2. The InfoPad configuration differs from traditional direct-sequence spread spectrum schemes in several respects. As usual, each user data bit is mapped into N chips where N is the spreading factor. In the past, pseudonoise (PN) sequences or Gold codes -- two types of nearly orthogonal codes -- have been used to spread as well as multiplex the users. These codes have been

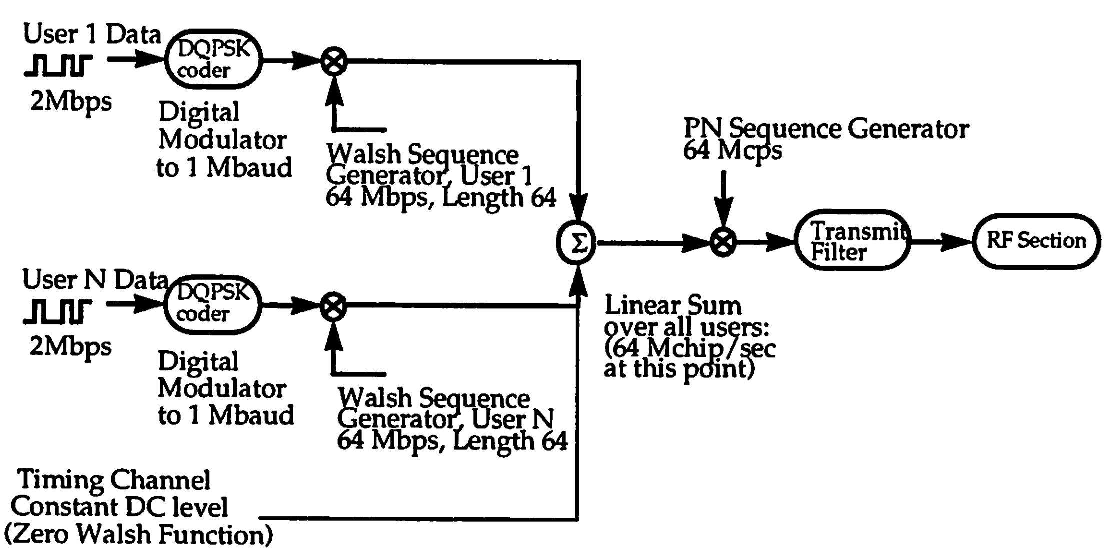
User 1 Data
2Mbps
DQPSK coder
Digital Modulator to 1 Mbaud
Walsh Sequence Generator, User 1 64 Mbps, Length 64
User N Data
2Mbps
DQPSK coder
Digital Modulator to 1 Mbaud
Walsh Sequence Generator, User N 64 Mbps, Length 64
Timing Channel Constant DC level (Zero Walsh Function)
Σ
Linear Sum over all users: (64 Mchip/sec at this point)
PN Sequence Generator 64 Mcps
Transmit Filter
RF Section

FIGURE 2.1 : InfoPad Downlink Transmitter Block Diagram

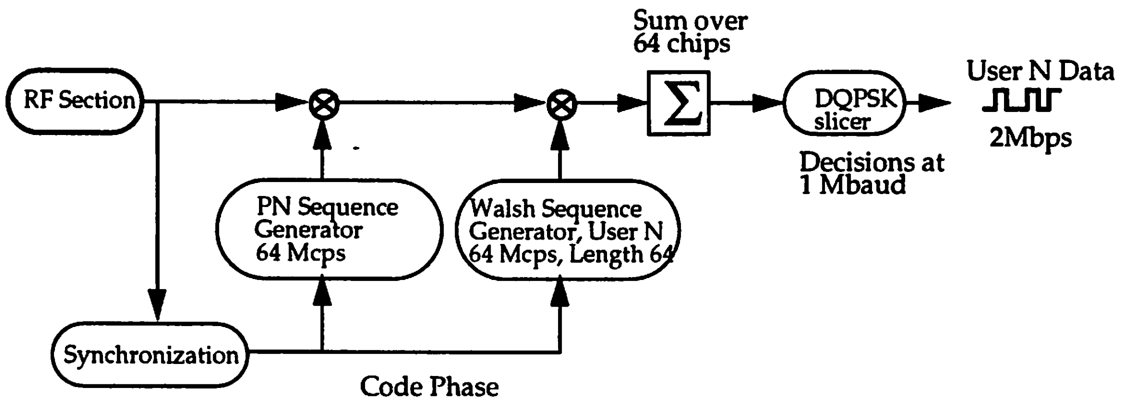

FIGURE 2.2 : InfoPad Mobile Receiver Block Diagram

studied extensively in the literature and an excellent treatment of their properties can be found in [Holmes82]. In the InfoPad system however, users are assigned codes from the set of Walsh sequences, a class of perfectly orthogonal codes that are easy to generate. Although these sequences have very poor autocorrelation properties and spectral characteristics, they permit up to N users to be multiplexed with a spreading factor of only N, a significant advantage over the more popular codebooks. Three examples of Walsh code spreading sequences are shown in Figure 2.3 for a spreading factor of 64. A treatment of Walsh sequences, both from a theoretical and practical viewpoint can be found in [Beau75]. It should be noted that the zero Walsh function, a DC signal, is used as a pilot channel for timing recovery in the InfoPad. This permits a receiver, once synchronized to the base station, to easily demodulate any of the user data channels without additional synchronization.

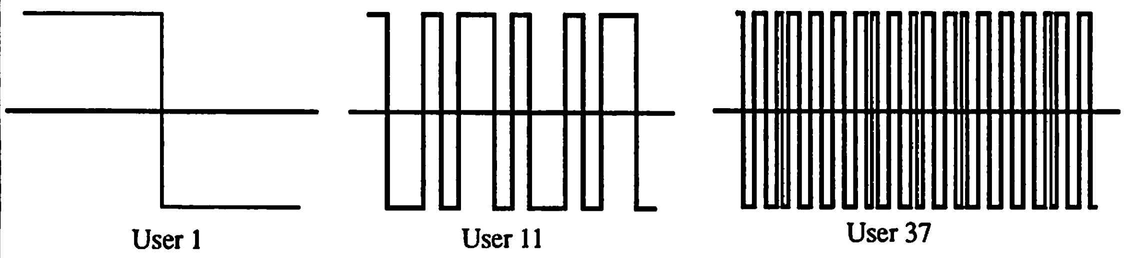

FIGURE 2.3 : Walsh Functions

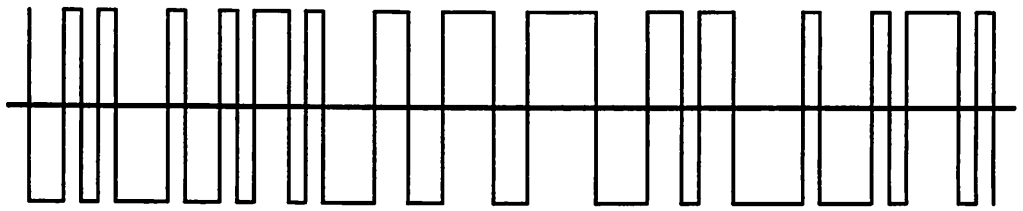
FIGURE 2.4 : PN Code Segment

2. 1. 2 Signal Autocorrelation Properties

To preserve the desirable autocorrelation properties normally associated with a direct sequence spread spectrum (DS-SS) system, the InfoPad system maps a long PN sequence one-to-one with chips in the multiplexed stream before filtering, modulation, and transmission. This mapping effectively "scrambles" the data stream, causing it to have the noise-like properties for which PN codes have been employed traditionally in DS-SS implementations. A segment of one such long PN scrambling code is shown in Figure 2.4. Figure 2.5 illustrates two autocorrelation functions for PN sequences. The first is a traditional PN code of length 127; the second has an additional bit added to aid in the InfoPad system implementation. These autocorrelation functions are, of course, periodic with the code length, although the periodicity is not shown explicitly here. At the receiver, a timing recovery circuit synchronizes the local PN sequence generator to the incoming signal and "unscrambles" it, restoring the original multiplexed data. Individual user bits may then be recovered by correlating the multiplexed stream with

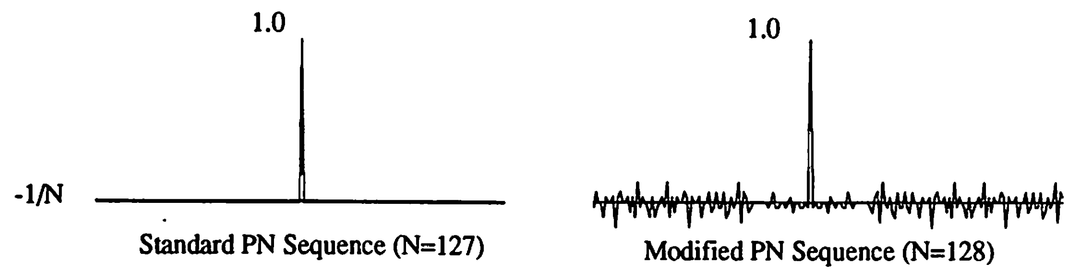

FIGURE 2.5 : PN Autocorrelation Functions

the Walsh code sequence assigned to the desired user. Since the Walsh codes are perfectly orthogonal, this operation selects exactly one user stream (in the absence of channel effects). Note that this correlation is performed only over N chips where N is the spreading factor. The length of the PN code has no influence on this correlation length.

Preliminary research has selected a spreading factor of 64 for the InfoPad project, with a proposed data rate of 2Mbps for each individual user. Using DQPSK at a symbol rate of 1 Mbaud implies a chip rate of sixty four million chips per second, or a chip period of approximately 16 ns. A PN code length of 32,768 has been chosen. Carrier frequencies will be centered around 1.5 GHz. For a more detailed description of factors influencing these decisions, see [Sheng91].

Also referenced and explained in [Sheng91] are the statistical channel models used in this work (see also [Saleh87]). The models are semi-empirical, based both on measurements of the indoor channel and theoretical models. A Rician distribution for delayed path arrivals is assumed. The absolute magnitude impulse response of one statistical channel profile is illustrated below in Figure 2.6. Multipath arrivals of non-negligible amplitude arrive up to 90 ns after the original line-of-sight arrival. On average, for these channels delay spreads range from 20 to 50 ns, corresponding to two

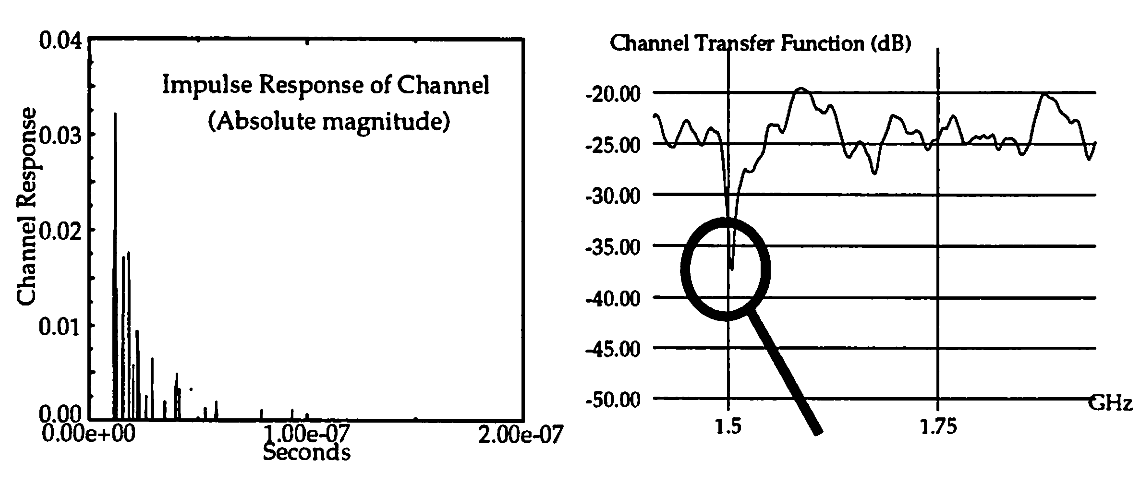

FIGURE 2.6 : Impulse Response of Typical Channel

or three chip times. Also shown is the magnitude frequency response of this particular channel, with the notch near 1.5 GHz denoting frequencies in a deep fade.

To combat such fades, the spread signal is designed to have a bandwidth much larger than the coherence bandwidth of the channel. Spreading the information content over a bandwidth wider than the coherence bandwidth provides several independent fading paths for the receiver. The probability that frequencies separated by more than the coherence bandwidth are both in a fade is small. In this sense, spread spectrum is simply a form of frequency diversity [Proak89]. For these channels, the coherence bandwidth is between 20 and 50 MHz and the InfoPad spread bandwidth is 64 MHz. If the channel models accurately represent the indoor environment, the proposed spreading factor should be sufficient. As with most of the quantitative results in this paper, however, this conclusion depends substantially on the accuracy of the models.

3 CDMA Synchronization

Timing recovery in a direct-sequence spread spectrum system comprises one of the most critical and difficult aspects of receiver design. Because of the sharply peaked autocorrelation properties of the PN sequences used as scrambling codes, the local clock at the receiver must remain synchronized to within a phase error of less than one half chip time for effective extraction of information from the incoming signal. With InfoPad chip times on the order of 16 ns, the task is non-trivial. The problem is made more daunting by the presence of carrier modulation. The difficulty of realizing coherent demodulation at these frequencies and chip rates necessitates the use of differential quadrature phase shift keying (DQPSK) which permits incoherent detection. No attempt is made to phase lock the incoming signal, and the conversion to baseband is performed by a sampling demodulator. Details may be found in [Sheng91]. Incoherent detection makes the system robust against small frequency offsets between carrier and receiver. Consequently, for this report, carrier recovery issues are neglected and it is assumed that perfect mixing down to baseband is provided. Hereafter, focus will be directed toward timing recovery techniques which may be implemented digitally at baseband. This chapter discusses the issues of timing recovery studied in simulation.

3.1 Timing Recovery: Two Major Tasks

Spread spectrum timing recovery has typically been divided into two tasks: acquisition and tracking. Acquisition involves coarse alignment of signal and local PN sequences to within one half a chip time. Once acquisition is achieved, control passes to the tracking circuity where still finer timing alignment is attained and maintained in the presence of jitter, noise, fading, multipath signal degradation, and other deleterious channel effects. If synchronization is lost at any time, the tracking system must recognize this condition and transfer control back to the acquisition portion of the circuit. Because acquisition in a DS-SS system is typically slow and computationally intensive, it is desirable to perform acquisition as infrequently as possible. In addition, acquisition search time results in lost data unless the transmitter has knowledge of the situation and substitutes a training sequence for data. In contrast, a tracking error degrades receiver SNR but still may permit data recovery. Hence tracking circuitry with a wide "pull-in" range is desired, even at the expense of slower responsivity.

3. 1. 1 Acquisition

In an effort to solve the acquisition problem, numerous approaches have been investigated, ranging from simple exhaustive serial searches, to parallel searches, partial and pattern searches, and even complicated sequential estimation algorithms. A simple calculation underscores the computational intensiveness and latency impact the acquisition algorithm has the potential to create:

For the InfoPad project, a PN code of length 2^{15}=32768 has been proposed. The chip rate for a spreading factor of sixty-four is 64 Mchips/sec. Suppose the incoming signal is sampled at twice the chip rate (in order to provide alignment to within one half a chip time), and an exhaustive serial search is performed. Let the samples be labeled $x_{nm}(t)$ where $x_{nm}(t)=x(nT_c+ mT_c/2)$, $m\in \{0,1\}$, and let PN_k represent the periodic PN code ($PN_k \in \{-1,+1\}$). For each phase offset, a digital correlation is performed:

$$C_m(k) = \sum_{n=0}^{N} x_{nm} PN_{n+k}$$

and the maximum $c_m(k)$ is selected as the proper alignment phase. Assume the correlation is implemented by means of a digital filter sampled at $t=NT_c$, and that the receiver employs a single correlator, correlating over the entire sequence length. Each correlation will require NT_c= (32768 chips)(15 nsec/chip) = 512 μs. There are (32768 *2 = 65536) such correlations to perform since we desire accuracy to within $T_c/2$, so the entire serial search will require 33.55 seconds for correlation alone, neglecting processing! If synchronization is ever lost during transmission, a similar waiting period would be required before reliable transmission could resume.

Good general discussions of the acquisition problem may be found in [Davies73, Pick82]. Recent work to reduce acquisition time is contained in [Subr91], while a new and novel approach is suggested in [Bree91]. Neither approach was employed in the Infopad system. The first was dismissed for complexity reasons; the second, because the frequency fading channel characteristics may render the technique ineffective. Instead, the InfoPad proposal utilizes the simplest technique (from the standpoint of complexity) employing four parallel correlators to conduct an exhaustive search on all code phase offsets. During acquisition mode operation, all four correlators work in unison to find the strongest signal, thereby reducing the search time approximately fourfold. A partial correlation of only 1024 chips is performed in acquisition mode to reduce the integration interval. Because the base station boosts the relative power in the pilot signal compared to other users, correlation against the pilot signal results in large energy detection once the local sequence is properly aligned.

Unfortunately, in simulation the acquisition operation can introduce exorbitant delays since parallelism cannot be easily exploited. In fact, as the PN scrambling sequence length grows much longer than 1024 chips, acquisition search time in simulation becomes unreasonably lengthy. For this reason, PN codes of length 256 were typically used to expedite simulations. Short PN sequences should not have a noticeable impact on the system's simulated performance, since received data bits are not correlated over the entire length of the scrambling code in the InfoPad approach. Orthogonality is preserved by the Walsh functions, and bit correlation is performed only over the number of chips equal to the spreading factor. As long as the PN scrambling code is longer than the spreading factor, there is no further immunity to multipath effects with a longer code. Only one motivation warrants the use of long codes at all: the original IS-95 standard uses a long code to differentiate base stations by their code phase offsets, allowing a synchronized user to identify separate base station signals and multipath reflections, and helping simplify the handoff mechanism. Since the present simulation work examines only transmission from a single base station to multiple users, the use of shorter PN codes to speed simulations should not impact results in any significant way.

3.1.2 Tracking

Once coarse acquisition is achieved, still finer synchronization is achieved by the tracking circuitry. Tracking methods in a DS-SS system have been widely studied, and the literature is easily accessible [Holmes82, Davies73, Proak89]. By far the most popular technique is the delay-locked loop (DLL), or minor variants of it [Yost82]. Conceptually, the technique is quite simple and intuitive. Two correlators, termed "early" and "late" for reasons that will become obvious, are employed in tracking while

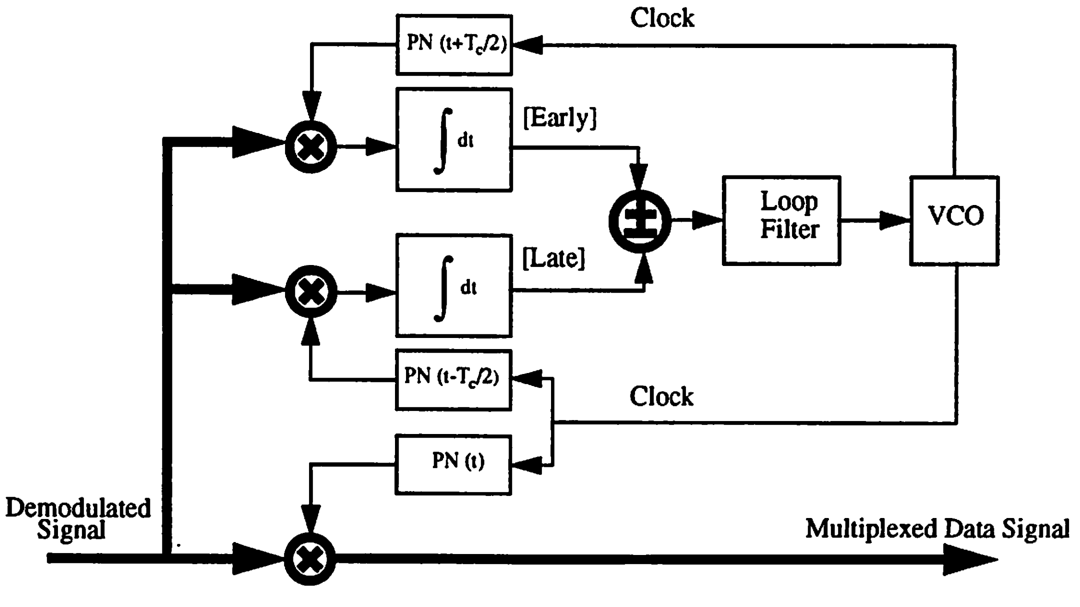

FIGURE 3.1 : Delay Locked Loop Block Diagram

a third is used for the extraction of data. The early correlator correlates with a PN sequence that is advanced by some fraction of a chip period (typically one half chip time), while the late correlator utilizes a sequence delayed by the same fraction. A block diagram of the DLL is shown in Figure 3.1.

A difference signal is formed by subtracting the late correlation output from the early. This difference signal, filtered and multiplied by an appropriate gain value is used in a feedback loop to drive the local clock. It can be verified by inspection that when the late correlation output value exceeds the early one, a signal is generated to advance the clock since the receiver is sampling too late. Similarly, when early exceeds late, the clock is delayed. When the early and late correlations are equal (assuming the correlation curve to be symmetric) the difference signal is zero, and the clock phase remains fixed since the ontime correlator is now sampling the signal at the correlation peak. A single correlator output as a function of code phase for values near alignment is shown in Figure 3.2. This figure also shows the difference signal as a function of code phase offset. It should be noted that these correlation curves were generated by the system illustrated in Figure 3.3 where a pilot channel only is encoded, the transmit filter is a raised cosine with 50% excess bandwidth, and an impulsive channel is inserted.

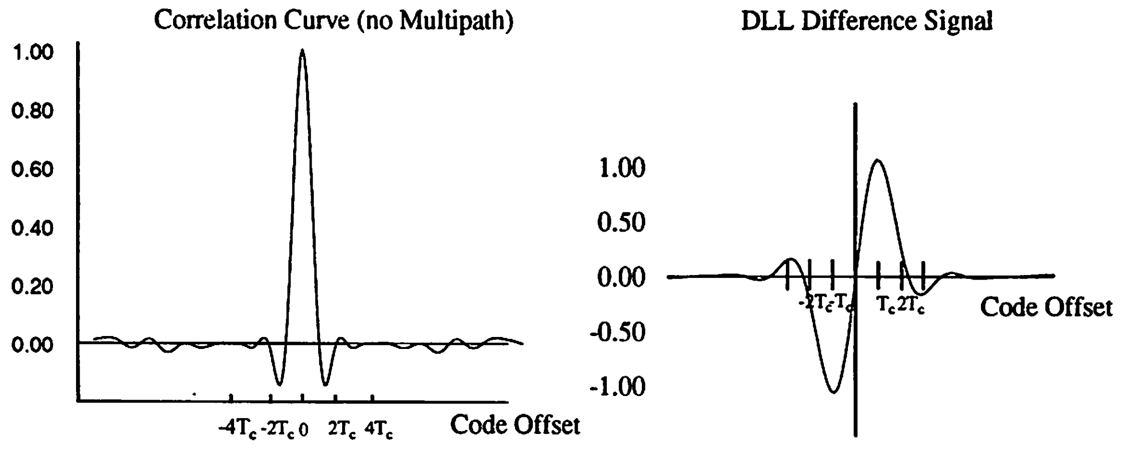

FIGURE 3.2 : DLL Correlation Curve and Difference Signal

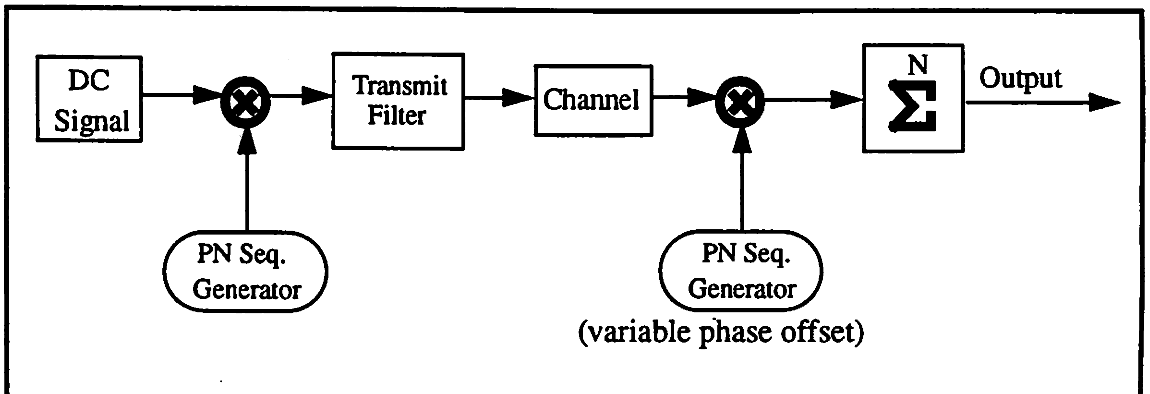

FIGURE 3.3 : System to Generate Correlation Curve

Proper operation of the DLL depends, to some extent, on the symmetry of the correlation curve. Introduction of a multipath channel distorts this curve. Figure 3.4 illustrates a real-valued correlation curve in the presence of multipath interference for comparison. The presence of delayed, superposed echoes on the pilot channel is apparent. As mentioned above, typical channel delay spreads are 20-50 ns with a chip rate of approximately 15 ns, explaining the absence of distinct peaks. This "blurring" makes independent tracking of echoed paths infeasible, as will be discussed later. Fortunately, one of the important conclusions of this work is to establish acceptable DLL performance despite the presence of asymmetrical correlation curves. A simple DLL tracking technique should provide adequate performance for the InfoPad receiver.

3. 1. 3 Loss of Lock Detection

Because both the early and late correlators are spaced only one half chip time away from the peak of the correlation curve, both receive significant signal energy

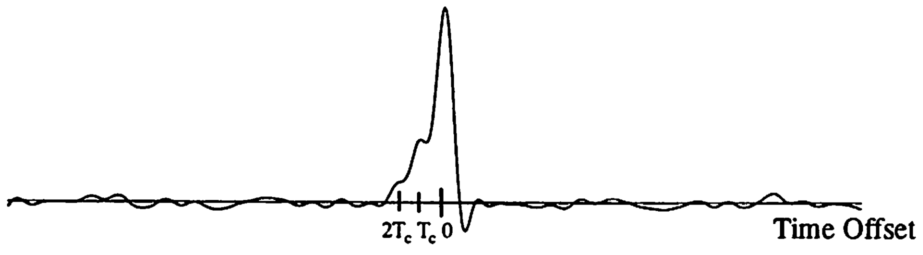

FIGURE 3.4 : Correlation curve with multipath

when the local PN sequence generator is operating in lock with the incoming signal. Hence, while the difference in the two values is used to drive the sampling phase loop, the sum can be applied to a threshold device to ascertain that the receiver lock is maintained. If the test fails, control is returned to the acquisition portion of the timing recovery system.

3. 1. 4 Feedback Filter Considerations

One final consideration that has not been discussed here is the use of a filter in the feedback path of the sampling phase adjust. A simple gain element functioned adequately for purposes of this work. A more detailed analysis of the loop filter should be included in future work when frequency offset effects are considered.

4 Interference and Diversity

The InfoPad downlink is designed to function as an interference-limited communication system. This means that interference, both self-induced and from other users, is the primary factor influencing signal degradation at the receiver through the introduction of noise. Thermal effects are sufficiently small relative to the interference noise that they can be normally neglected in the analysis. Since InfoPad uses Walsh functions to multiplex users in an orthogonal fashion, the sources of interference may not be clear, particularly when transmission from a single base station alone is being considered. It is the multipath characteristic of the link which introduces the interference effects. Because of reflections from ceilings, walls, floors, people, and other objects in a room, there is attenuation and multiple path delay in the arriving signal. As a result, the receiver at any given instant sees not only the desired signal but a number of echoes from previous signals from reflected paths. Path delays larger than the symbol rate give rise to intersymbol interference (ISI); more closely spaced arrivals are termed intrasymbol or interchip interference. As noted previously, in an indoor environment at these bit rates, the latter effect is dominant and can be quite severe.

Unfortunately, in interference-limited situations, if the receiver signal to noise ratio (SNR) is not large enough to insure reliable communication, the transmitter cannot rectify the problem by simply increasing transmit signal power. Any increase in signal power results in a proportional increase in noise power, since interference scales proportionally to signal. For the same reason, additional users in a multipath environment, even if they are perfectly orthogonally multiplexed, can contribute to system interference. The system designer must either mitigate the impact of multipath arrivals, increase the spreading factor, or reduce the number of users. The following analysis attempts to quantify the effects of user interference in the InfoPad system configuration. The analysis follows closely work done in [Cout94].

4.1 Multipath and Interference

The system under consideration employs direct-sequence spread spectrum for transmission. Since the fading channel can be resolved only at multiples of the chip time T_c, a discrete time impulse response model for the channel is used

$$h(t) = \sum_{l=0}^{L-1} \beta_l \delta(t - lT_c) e^{j\theta_l} \qquad \text{(EQ 4.1)}$$

where L is the number of resolvable multipaths, $\{\beta_l\}$ are random amplitudes, and $\{\theta_l\}$ are random phases. Consider a base station transmitting a direct sequence (DS) spread spectrum DQPSK signal to K mobile units in parallel. The passband signal a(t) transmitted by the base station is the superposition of K signals intended for the different users

$$a(t) = \sum_{k=1}^{K} d_k(t-\tau_k) p_k(t-\tau_k) e^{j(2\pi f_0 t + \phi_k)} \qquad \text{(EQ 4.2)}$$

where $d_k(t)$ is the kth user's data signal, $p_k(t)$ is the spreading sequence for the kth user, ϕ_k is the random phase of the kth carrier, and τ_k is the random time delay. Assume the base station transmits all signals bit and carrier synchronously so that the delay τ_k and the phase ϕ_k are identically zero for each of the K signals. Each data signal is modeled as

$$d_k(t) = \sum_{m=-\infty}^{\infty} b_m^k \cdot P_T(t-mT) \qquad \text{(EQ 4.3)}$$

where b_m^k represents the kth user's differentially encoded QPSK symbol, chosen from the set $\{e^{j\theta}: \theta \in \pi/4, 3\pi/4, 5\pi/4, 7\pi/4\}$, T is the bit period and $P_T(\cdot)$ represents a rectangular pulse with amplitude 1 and width T. The spreading code consists of a sequence of rectangular chips of duration T_c taking on values x_m^k from the set $\{\pm 1\}$:

$$p_k(t) = \sum_{m=-\infty}^{\infty} x_m^k P_{T_c}(t-mT_c) \qquad \text{(EQ 4.4)}$$

where $P_{T_c}(*)$ represents a unit amplitude pulse with width T_c.

The superimposed signals all fade in unison over the same channel, so the signal received at the terminal is

$$r(t) = a(t) * h(t) = \sum_{l=0}^{L-1} \sum_{k=1}^{K} \beta_l d_k(t-lT_c) p_k(t-lT_c) e^{j(2\pi f_0(t-lT_c)+\theta_l)} \qquad \text{(EQ 4.5)}$$

This signal may be decomposed into a superposition of signals: one representing the reference user's signal and self-interference, and a second representing the combined interference of the signals transmitted to other mobile users. With the reference user arbitrarily assigned to user 1, this decomposition is

$$\begin{aligned} r(t) &= \sum_{l=0}^{L-1} \beta_l d_1(t-lT_c) p_1(t-lT_c) e^{j(2\pi f_0(t-lT_c)+\theta_l)} \\ &\quad + \sum_{k=2}^{K} \sum_{l=0}^{L-1} \beta_l d_k(t-lT_c) p_k(t-lT_c) e^{j(2\pi f_0(t-lT_c)+\theta_l)} \end{aligned} \qquad \text{(EQ 4.6)}$$

For simplicity of notation, define two partial correlation functions

$$R_{jk}(\tau) = \frac{1}{T}\int_0^{\tau} p_j(t-\tau) p_k(t)\, dt \qquad \text{(EQ 4.7)}$$

$$\hat{R}_{jk}(t) = \frac{1}{T}\int_{\tau}^{T_c} p_j(t-\tau) p_k(t)\, dt \qquad \text{(EQ 4.8)}$$

Consider an integrate-and-dump receiver employing DQPSK to recover the desired bit. The decision variable is formed by multiplying the output of the correlator at time t by the conjugate of the output at time (t-T). The result is applied to a QPSK slicer to determine the data bit. The output of the correlator at the first bit time is

$$g(T) = \frac{2}{T} \cdot \int_0^T r(t) p_1(t) e^{-j2\pi f_0 t} dt \qquad \text{(EQ 4.9)}$$

$$
\begin{aligned}
g(T) &= \beta_0 b_0^1 e^{j\theta_0} \\
&+ \sum_{l=1}^{L-1} \beta_l e^{j(-2\pi f_0 l T_c + \theta_l)} \int_0^T d_1(t - lT_c) p_1(t)\, dt \\
&+ \sum_{l=0}^{L-1} \sum_{k=2}^{K} \beta_l e^{j(-2\pi f_0 l T_c + \theta_l)} \int_0^T d_k(t - lT_c) p_1(t)\, dt
\end{aligned}
\qquad \text{(EQ 4.10)}
$$

where we use the fact that

$$
\frac{2}{T} \cdot \int_0^T \beta_0 d_1(t) p_1^2(t) e^{j\theta_0} dt = \beta_0 b_0^1 e^{j\theta_0} \qquad \text{(EQ 4.11)}
$$

In Equation 10 above the first term represents the signal component, while the second and third terms represent self- and other-user interference, respectively. Note that d(t) changes value from b_{-1}^k to b_0^k at t=0. Consequently, we can rewrite the integral expressions in terms of the partial correlation functions previously defined. Namely

$$
\begin{aligned}
g(T) &= \beta_0 b_0^1 e^{j\theta_0} \\
&+ \sum_{l=1}^{L-1} \sum_{k=1}^{K} \beta_l e^{j(-2\pi f_0 l T_c + \theta_l)} \left[b_{-1}^k R_{k1}(lT_c) + b_0^k \hat{R}_{k1}(lT_c) \right] \\
&+ \beta_0 e^{j\theta_0} \sum_{k=2}^{K} \left[b_{-1}^k R_{k1}(0) + b_0^k \hat{R}_{k1}(0) \right]
\end{aligned}
\qquad \text{(EQ 4.12)}
$$

Terms are regrouped here for simplicity. In the second term, note the presence of both the "present" symbol b_0^k as well as the "previous" symbol b_{-1}^k. This simply represents the fact that for delayed paths, a portion of the previous symbol will arrive in the correlator during the present correlation period. Since it is fairly easy to select orthogonal or nearly orthogonal codes which have small partial correlations at zero offset, the last term in this expression can be approximated as zero.

Summarizing, we may write

$$
g(T) = S_0 + I_0 \qquad \text{(EQ 4.13)}
$$

where the signal and interference are given by

$$S_0 = \beta_0 b_0^1 e^{j\theta_0}$$

$$I_0 = \sum_{l=1}^{L-1} \sum_{k=1}^{K} \beta_l e^{j(-2\pi f_0 l T_c + \theta_l)} \left[b_{-1}^k R_{k1}(lT_c) + b_0^k \hat{R}_{k1}(lT_c) \right] \quad \text{(EQ 4.14)}$$

In a similar fashion, the demodulator output at the second bit time will then be

$$g(2T) = S_1 + I_1 \quad \text{(EQ 4.15)}$$

with

$$S_1 = \beta_0 b_1^1 e^{j\theta_0} \quad \text{(EQ 4.16)}$$

$$I_1 = \sum_{l=1}^{L-1} \sum_{k=1}^{K} \beta_l e^{j(-2\pi f_0 l T_c + \theta_l)} \left[b_0^k R_{k1}(lT_c) + b_1^k \hat{R}_{k1}(lT_c) \right] \quad \text{(EQ 4.17)}$$

The DQPSK decision variable is

$$Y = g(2T) g^*(T) = \left(S_1 + I_1\right)\left(S_0^* + I_0^*\right) = S_1 S_0^* + I_1 S_0^* + S_1 I_0^* + I_1 I_0^*$$
$$= \beta_0^2 b_1^1 b_0^{1*} + \beta_0 b_0^{1*} e^{-j\theta_0} I_1 + \beta_0 b_1^1 e^{j\theta_0} I_0^* + I_1 I_0^* \quad \text{(EQ 4.18)}$$

The first term is obviously the desired signal

$$S = \beta_0^2 b_1^1 b_0^{1*} = \beta_0^2 s_1^1 \quad \text{(EQ 4.19)}$$

where s_1^1 represents the first user's first data symbol (DQPSK encoding removed). The other three terms in the expression for Y constitute interference

$$I = \beta_0 b_0^{1*} e^{-j\theta_0} I_1 + \beta_0 b_1^1 e^{j\theta_0} I_0^* + I_1 I_0^* \quad \text{(EQ 4.20)}$$

We will neglect the last term as small relative to other terms in the expansion, and approximate the remaining terms as a zero-mean complex Gaussian random variable. A simple computation shows that the real and imaginary parts of this random variable are uncorrelated and thus independent. Here we will compute the variance of the real portion

$$VAR\,[Re(I)] = VAR\left(\frac{1}{2}[I+I^*]\right) = \frac{1}{2}E\left[(I+I^*)^2\right] = \frac{1}{2}E\,[II+2II^*+I^*I^*]$$

$$\begin{aligned}
&= \frac{\beta_0^2}{2}E\Big[\left(b_0^{1*}\right)^2 e^{-j2\theta_0} I_1^2 + 2b_0^{1*}b_1^1 I_1 I_0^* + \left(b_1^1\right)^2 e^{j2\theta_0}\left(I_0^*\right)^2 \\
&\quad + 2|I_1|^2 + 2b_0^{1*}b_1^{1*}e^{-j2\theta_0} I_1 I_0 + 2b_0^1 b_1^1 e^{j2\theta_0} I_1^* I_0^* + 2|I_0|^2 \qquad \text{(EQ 4.21)} \\
&\quad + \left(b_0^1\right)^2 e^{j2\theta_0}\left(I_1^*\right)^2 + 2b_0^1 b_1^{1*} I_1^* I_0 + \left(b_1^{1*}\right)^2 e^{-j2\theta_0} I_0^2\Big]
\end{aligned}$$

$$= \frac{\beta_0^2}{2}E\left[2b_0^{1*}b_1^1 I_1 I_0^* + 2|I_1|^2 + 2|I_0|^2 + 2b_0^1 b_1^{1*} I_1^* I_0\right]$$

where the last step follows from the fact that θ_0 is independent of I_0, I_1, and the data sequence. Furthermore, it is uniformly distributed on $[0,2\pi]$ causing the expectation of terms containing θ_0 to go to zero. The independence of all user bit streams and all phase angles $\{\theta_l\}$ allows us to write

$$E\left[b_0^{1*}b_1^1 I_1 I_0^*\right] = E\left[b_0^1 b_1^{1*} I_1^* I_0\right] = 0 \qquad \text{(EQ 4.22)}$$

implying

$$VAR\,[Re(I)] = \beta_0^2 E\left[|I_1|^2 + |I_0|^2\right] \qquad \text{(EQ 4.23)}$$

We may also observe that

$$E\left[|I_0|^2\right] = E\left[|I_1|^2\right] = \sum_{l=1}^{L-1}\sum_{k=1}^{K} E\left[\beta_l^2\right]E\left[R_{k1}^{\;2}(lT_c) + \hat{R}_{k1}^{\;2}(lT_c)\right] \qquad \text{(EQ 4.24)}$$

To compute this last expectation requires knowledge of the user codes. Although in practice these codes are deterministic, a reasonable approximation for DS-SS systems is to assume a random code where all code elements x_j^k are independent, identically distributed Bernoulli random variables having equal probability of being positive or negative one. Using this approximation, we can compute

$$\begin{aligned}
E\left[R_{k1}^2\left(lT_c\right)\right] &= E\left[\left(\frac{1}{T}\cdot\int_0^{lT_c} p_k\left(t-lT_c\right)p_1(t)\,dt\right)\left(\frac{1}{T}\cdot\int_0^{lT_c} p_k\left(\tau-lT_c\right)p_1(\tau)\,d\tau\right)\right] \\
&= \frac{1}{T^2}E\left[\left(\sum_{i=0}^{l-1} T_c x_i^1 x_{i-l}^k\right)\left(\sum_{j=0}^{l-1} T_c x_j^1 x_{j-1}^k\right)\right] \qquad \text{(EQ 4.25)}
\end{aligned}$$

$$E\left[R_{k1}^2\left(lT_c\right)\right] = \frac{T_c^2}{T^2}E\left[\left(\sum_{i=0}^{l-1}\sum_{j=0}^{l-1}\delta(i-j)\right)\right] \tag{EQ 4.26}$$
$$= \frac{l}{N^2}$$

A similar computation shows

$$E\left[R_{k1}^2\,(lT_c)\right] = \frac{(N-l)}{N^2} \tag{EQ 4.27}$$

This yields a final result for the interference term given by

$$VAR\,[Re\,(I)\,] = 2\beta_0^2\sum_{l=1}^{L-1}\sum_{k=1}^{K}E\left[\beta_l^2\right]\left(\frac{l}{N^2}+\frac{(N-l)}{N^2}\right)$$
$$= 2\beta_0^2\cdot\frac{K}{N}\cdot\sum_{l=1}^{L-1}E\left[\beta_l^2\right] \tag{EQ 4.28}$$

In the same manner, it is easily shown that

$$VAR\,[Im\,(I)\,] = Var\,[Re\,(I)\,] = 2\beta_0^2\cdot\frac{K}{N}\cdot\sum_{l=1}^{L-1}E\left[\beta_l^2\right] \tag{EQ 4.29}$$

If we divide both signal and interference powers by $\beta_0{}^2$ to normalize, we arrive at the result

$$VAR\,[Im\,(I)\,] = Var\,[Re\,(I)\,] = 2\cdot\frac{K}{N}\cdot\sum_{l=1}^{L-1}E\left[\beta_l^2\right] \tag{EQ 4.30}$$

In summary, the interference term can be approximated as a zero mean complex Gaussian random variable with independent real and imaginary components, each having variance equal to that given by Equation 4.30. An examination of this variance result confirms our original intuition. Interference in the system is proportional to the number of users K and inversely proportional to the spreading factor N. Moreover, the sum of the magnitude squared of all path delay ($0<l<L$) coefficients is proportional to the total interference in the system. This last term may be viewed as the magnitude of the channel dependent interference for a simple multipath rejection receiver which attempts no interference cancellation or ratio combining.

Under these conditions, the in-phase and quadrature channels function independently and each has a signal to noise ratio γ given by

$$\Upsilon = \frac{N\beta_0^2}{2K \cdot \sum_{l=1}^{L-1} E\left[\beta_l^2\right]} \qquad \text{(EQ 4.31)}$$

The approximate bit error rate of the link is given in [Proak89] as:

$$P_e = erfc\,(\sqrt{\Upsilon}) \cdot \left[1 - \frac{1}{4} erfc\,(\sqrt{\Upsilon})\right] \qquad \text{(EQ 4.32)}$$

4.2 Diversity Combining

It is clear that as the number of users in a single cell area increases to the maximum allowed (the spreading factor), interference may swamp the signal, particularly if there are significant amplitude reflections. Diversity combining is often proposed as a method to combat this problem. Diversity can be exploited in numerous ways: time diversity can be achieved by interleaving and forward error correction (FEC) at the base station transmit end; antenna diversity in the indoor environment has already been proposed [Camag93], and frequency diversity is inherent in the spread-spectrum approach, as previously discussed.

4. 2. 1 RAKE Receiver

Another potentially useful form of diversity is the multipath nature of the signal itself. Signals transmitted from the base station arrive at the receiver with various delays. The presence of a pilot tone at the receiver provides a reasonable estimate of the channel delay profile. Conceptually, delayed copies of the signal arriving at the receiver might be combined in some optimal fashion to increase the SNR at the decision threshold. In essence, the idea is to gather signal energy over several fading paths and recombine the multiple receptions coherently. A detailed discussion of multipath diversity combining possibilities is discussed in [Holmes73, Proak89]. The most popular proposal for diversity combining is the well-known RAKE receiver structure. The optimal RAKE receiver employs independent receivers (often termed "fingers") to track and demodulate independent delay paths. For the InfoPad system, this approach is not feasible since low-pass filtering at the channel bandwidth blurs multipath arrivals together, as illustrated by the multipath correlation curve shown in chapter three. One obvious compromise is to construct a modified RAKE receiver whose fingers are arbitrarily spaced one chip time apart. The resulting structure is shown in Figure 4.1.

It is well known that in the presence of fixed-level Gaussian white noise, the RAKE receiver provides the optimal solution to the diversity combining problem. Simply put, under these conditions signals are combined with a weight proportional to the amplitude of their signal component. The resultant decision variable has a signal-to-noise ratio equal to the sum of the individual SNR's of the separate path receivers.

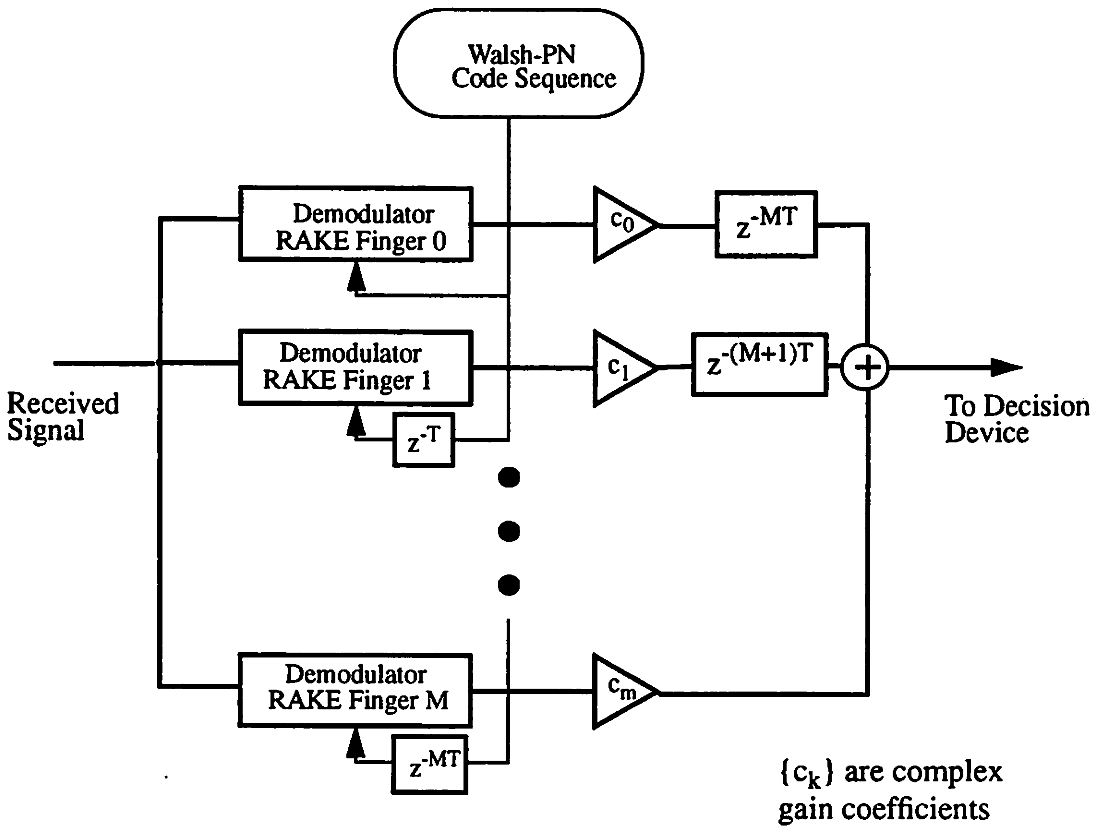
Walsh-PN Code Sequence
Demodulator RAKE Finger 0
Demodulator RAKE Finger 1
Demodulator RAKE Finger M
c_0
c_1
c_m
z^{-MT}
$z^{-(M+1)T}$
z^{-T}
z^{-MT}
Received Signal
To Decision Device
$\{c_k\}$ are complex gain coefficients

FIGURE 4.1 : Modified RAKE receiver structure

Several recent proposals have investigated extension of the RAKE receiver to an interference-limited scenario [Grob90, Nore94].

4. 2. 2 Impact of Interference-Limited Scenario

Unfortunately, the critical assumption employed in traditional derivations of the RAKE receiver -- fixed level background noise -- is not satisfied in an interference-limited environment, such as that envisioned for InfoPad. In this context, the preceding analysis discovered that the interference power at the receiver is a strong function of delayed path amplitudes. Indeed, for an arbitrary finger of the RAKE receiver, the interference variance term is proportional to the magnitude squared of all <u>other</u> resolvable path delays. In other words, if I_m is the zero-mean complex interference term for a receiver code-synchronized to the nth resolvable path, it is easily shown that

$$VAR[Re(I_m)] = VAR[Im(I_m)] = 2 \cdot \frac{K}{N} \sum_{l=0,\, l \neq m}^{L-1} E\left[\beta_l^2\right] \qquad \text{(EQ 4.33)}$$

This suggests that the variance of interference terms is not fixed for all path delay receivers. A simple example may clarify this crucial point. Suppose there are only two resolvable paths from transmitter to receiver, and the amplitude of the line-of-sight component is larger than the second resolvable path (i.e. $\beta_0 > \beta_1$). In the presence of fixed-level background noise (interference effects assumed negligible relative to thermal and other noises), a traditional RAKE receiver will combine signals from the first and second resolvable paths in the ratio β_0/β_1. When interference effects are not negligible, but dominate (as in InfoPad), this combining ratio must change. Intuitively, the receiver synchronized to the first resolvable path sees a signal proportional to β_0 and a noise proportional to β_1^2. Conversely, the second resolvable path receiver sees a signal proportional to β_1 and a noise term proportional to β_0^2. Clearly, the second receiver provides a much poorer estimate of the transmitted data in this case since it suffers from both a smaller signal component <u>and</u> a larger noise term. If noise terms at the two receivers are assumed independent (a somewhat pessimistic assumption since multiplication by the PN sequences shifted in time does not entirely uncorrelate the interferences) the new optimal combining ratio is easily computed.

4. 2. 3 Derivation of Optimal Ratio

Assume we have two received signals X_1 and X_2 related in the following fashion:

$$X_1 = S_1 + N_1 \qquad \text{(EQ 4.34)}$$
$$X_2 = ce^{j\theta}S_1 + N_2$$

where S_1 is the transmitted symbol. N_1 and N_2 are complex-valued Gaussian random variables with independent in-phase and quadrature components each having variance σ_1^2 and σ_2^2, respectively. The complex coefficient $ce^{j\theta}$ represents a channel perturbation of the second received signal relative to the first, and is assumed to be known.

The signaling method is assumed to be DQPSK and we wish to construct a linear combining strategy to minimize the BER at the receiver. We wish to construct a signal:

$$X_3 = X_1 + ae^{j\phi}X_2 = S_1\left(1 + ace^{j(\theta+\phi)}\right) + N_3 \qquad \text{(EQ 4.35)}$$

where BER is minimized by choice of $ae^{j\phi}$. Note that the composite noise term N_3 will be zero-mean, complex Gaussian with each component having variance $(\sigma_1^2 + a^2\sigma_2^2)$.

Assuming independent, equally likely transmission of symbols it is easy to show that maximizing the bit error rate is equivalent to maximizing (E_b/N_0) where N_0 represents the noise variance on each quadrature channel:

$$\frac{E_b}{N_0} = F\left(ae^{j\phi}\right) = \frac{|S_1|^2 \cdot \left|1 + ace^{j(\theta+\phi)}\right|^2}{\left(\sigma_1^2 + a^2\sigma_2^2\right)} \qquad \text{(EQ 4.36)}$$

$$F\left(ae^{j\phi}\right) = \frac{|S_1|^2 \cdot \left(1 + a^2c^2 + 2\cos(\theta+\phi)ac\right)}{\left(\sigma_1^2 + a^2\sigma_2^2\right)} \qquad \text{(EQ 4.37)}$$

From inspection, it is clear that maximizing this function over all choices of ϕ is accomplished by choosing $\phi=-\theta$. Additionally, the magnitude of the transmitted symbol does not affect the maximization and may be omitted. Hence we must maximize the following expression

$$F(a) = \frac{\left(1 + 2ac + a^2c^2\right)}{\left(\sigma_1^2 + a^2\sigma_2^2\right)} = \frac{(ac+1)^2}{\left(\sigma_1^2 + a^2\sigma_2^2\right)} \qquad \text{(EQ 4.38)}$$

Taking the derivative of this function with respect to a, setting it equal to zero and solving yields the value of a which maximizes this ratio

$$F'(a) = \frac{\left(\left(\sigma_1^2 + a^2\sigma_2^2\right)\cdot 2(ac+1)c\right) - \left((ac+1)^2\cdot 2a\sigma_2^2\right)}{\left(\sigma_1^2 + a^2\sigma_2^2\right)^2} = 0 \qquad \text{(EQ 4.39)}$$

$$\left(\sigma_1^2 + a^2\sigma_2^2\right)\cdot\left(ac^2 + c\right) = (ac+1)^2\cdot a\sigma_2^2 \qquad \text{(EQ 4.40)}$$

$$ac^2\sigma_1^2 + a^3c^2\sigma_2^2 + c\sigma_1^2 + a^2c\sigma_2^2 = a^3c^2\sigma_2^2 + 2a^2c\sigma_2^2 + a\sigma_2^2 \qquad \text{(EQ 4.41)}$$

$$a^2\left(c\sigma_2^2\right) + a\left(\sigma_2^2 - c^2\sigma_1^2\right) - c\sigma_1^2 = 0 \qquad \text{(EQ 4.42)}$$

$$a_{opt} = c\left(\frac{\sigma_1^2}{\sigma_2^2}\right) \qquad \text{(EQ 4.43)}$$

The optimal combining scheme in this case is then:

$$X_3 = X_1 + ce^{-j\theta}\left(\frac{\sigma_1^2}{\sigma_2^2}\right)X_2 \qquad \text{(EQ 4.44)}$$

4. 2. 4 Interpretation of Analysis

There are several interesting issues which this analysis raises. First, it is instructive to examine some specific cases. Revert to the scenario where there are two resolvable paths having amplitudes β_0 and β_1, and phase delays θ_0 and θ_1 respectively. and assume background noise is dominant so that the noise power at both receivers is assumed constant and uncorrelated. Consider a pre-detection RAKE combiner which attempts to linearly combine received signals from separate fingers tracking the zeroth and first path delay signals. The optimal solution can be written (incorporating a normalizing gain factor) as:

$$X_3 = \beta_0 e^{-j\theta_0}X_1 + \beta_1 e^{-j\theta_1}X_2$$

which is simply the matched filter solution. The matched filter is a term often given to the optimal receiver for a signal that has passed through a linear time-invariant (LTI) system in the presence of additive white Gaussian noise. If the impulse response of the channel is represented by [h_0, h_1, h_2, ..., h_n], the matched filter is the time-reversed conjugate of this impulse response. Intuitively, delayed versions of the desired signal weighted by delay-dependent amplitude coefficients appear at the receiver. It is a simple exercise to show that the total signal power in this composite signal is the sum of the squares of the tap coefficients. It is clear from above that the total signal energy in X_3 is simply $\beta_0^2+\beta_1^2$, as the matched filter solution predicts.

In the InfoPad system, interference -- not additive white noise -- is the dominant noise source. Noise powers for different RAKE fingers may not be equal because the noise arises from a signal-dependent interference term. Under these conditions it is clear that the more general expression for the combining ratio is required since σ_1^2 may not be equal to σ_2^2, in general. In particular, if the signal amplitudes on the line-of-sight and one-chip-delayed path are unequal, the noise (or interference) powers at the outputs of the two correlators will also be unequal.

Now consider the interference-limited scenario with path delay amplitudes of β_0 and β_1 and phase delays θ_0 and θ_1, as before. It is easy to show that the optimal linearly combined decision variable is:

$$Y' = Y_1 + \left(\frac{\beta_1}{\beta_0}\right) e^{-j(\theta_1-\theta_0)} \left(\frac{E[I_0^2]}{E[I_1^2]}\right) Y_2 = Y_1 + \left(\frac{\beta_1}{\beta_0}\right)^3 e^{-j(\theta_1-\theta_0)} Y_2 \qquad \text{(EQ 4.45)}$$

where the expressions for $E[I_0^2]$ and $E\{I_1^2]$ are given in Equations 4.24 and 4.28.

Note that the weighting coefficient for the second observable path receiver Y_2 increases as the cube of the amplitude ratio of the delay path amplitudes, rather than linearly. This has important ramifications for the combining operation. First, it is clear that if the amplitude of the direct path is even a factor of two larger than the delayed path, combining will yield little gain since the weighting coefficient will be 1/8. Since the indoor channels are characterized by a Rician distribution (implying a dominant line of sight component) addition of a RAKE receiver should provide little performance advantage. This conclusion is investigated in the following chapter.

5 Software Simulation of the InfoPad Downlink

With a basic understanding of the InfoPad system in place, the simulation work for this project may be readily understood. The Ptolemy platform, developed by students at UC Berkeley, was used as the primary software simulation tool. Early development was in Ptolemy's synchronous data flow (SDF) domain; for speed considerations many of the custom stars were ultimately migrated to the C code generation (CGC) domain. Eventually custom stars were written to model most of the elements in the downlink system described previously. Code for these stars is included (with comments) in Appendix A of this report. This chapter reviews code development briefly, presents simulation results, and attempts to establish links between theory presented previously and these simulations.

5.1 Ptolemy Simulation Development

5. 1. 1 Early Simulation Model

This research began as an extension of Paul Hurst's work on a delay-locked loop simulator [Hurst92]. Hurst developed a primitive Ptolemy model of the DLL in an effort to study the feasibility of simple timing recovery techniques at these chip rates. Hurst's model implemented a real-valued channel, and performed no coarse acquisition.

Proper operation required the user to manually calculate the delay through a fixed channel and explicitly set the appropriate PN code phase at the receiver. A new channel mandated a new code phase computation. PN sequences were used to multiplex users, and no effort was made to recover data bits at the receiver for BER simulation. Proper system operation was established by examining the sampler model output for settling to a fixed sampling phase; a crude power measurement at the output of the pilot channel correlator indicated receiver "lock". To investigate DLL tracking performance, Hurst used a chip rate channel model oversampled by a factor of ten. Linear interpolation provided a continuous sampling resolution adjustment for the DLL feedback signal.

5. 1. 2 Simulation Model Enhancements

Several improvements to this model were made. An acquisition star was written to automate system response to arbitrary filter delays, transient effects, and loss of lock scenarios. Provisions were made to allow dynamic switching between acquisition and tracking modes. Pilot channel power measurements automatically sensed loss of receiver lock and initiated acquisition operation. Since the indoor propagation model was implemented as a complex-valued baseband channel, it was also necessary to generate complex-valued equivalents for Hurst's primitive synchronization system. When the original QUALCOMM CDMA proposal became available in early 1993, a decision was made to study the performance of this modified system. A programmable, phase-variable Walsh sequence generator star was written, and the PN generator star was retained for use as a scrambler in the new implementation.

In the modified configuration, memory requirements for the original sampling star caused exorbitant delays and created deadlock conditions in Ptolemy's SDF domain. A more realistic sampling phase star was written that sampled at one half the chip period on each firing (instead of sampling an entire PN sequence length of chips in one firing). To speed simulation performance, acquisition and tracking blocks were combined into a single code block. Features were added to permit user selection of parameters such as search resolution, lock threshold, feedback gain, and acquisition integration length. In order to study the impact of various parameters on the bit error rate, adaptable AGC and A/D converter models were developed, as well as a bit error rate tester. The upgraded system was not only more versatile, but also executed five to eight times more quickly

This model functioned adequately for the investigation of synchronization and timing recovery issues. Results are presented in the following section. These indicate that a simple serial search acquisition and delay locked loop tracking should function adequately for the InfoPad system. With the primary goal of this research accomplished, it was decided to expand the investigation by considering bit error rates on the link and the parameters which affect its performance. Unfortunately, bit error rate measurements with this model required excessive computational time. A ten thousand bit simulation typically required fourteen hours to run! Two primary factors were believed to account for simulation length: excessive computation resulting from the oversampled channel and the overhead of working in Ptolemy's SDF domain.

Because the delay locked loop functioned so well, the sampling phase for a fixed channel settled after approximately 100 bits and remained constant throughout the rest of the simulation run. Except for the short transient period during which acquisition was performed and the DLL was settling, oversampling the channel was simply increasing the computational burden without providing much additional information. To rectify this problem, the settled phase value was recorded and a downsampled channel model was generated. Oversampled versions of the transmit filter, channel and receive filter were chained together and the composite impulse response was sampled at the chip rate with the appropriate sampling phase. Three oversampled filter blocks, acquisition and tracking models, and the front-end sampler were thus replaced by a single chip-rate filter. To further enhance performance, nearly all of the SDF stars were moved to Ptolemy's CGC domain. This domain generates a stand-alone C code simulator that can be separately compiled. The combination of these two changes now permits million bit simulations to run in less than an hour.

5. 1. 3 Simulation Model Applications

Presently these simulation models are being adapted by other UC Berkeley researchers to study issues including protocol development and forward error correction for a wireless link, power control trade-offs on the downlink, and frequency-hopped spread spectrum performance for the PATH project -- a low bit rate mobile vehicle communication system. Following is a summary of the simulation results for the issues considered in this project.

5.2 Simulation Results

5. 2. 1 Acquisition and Tracking

Acquisition and tracking performance of the DLL was evaluated by substituting several channels and executing the simulation to see whether the receiver

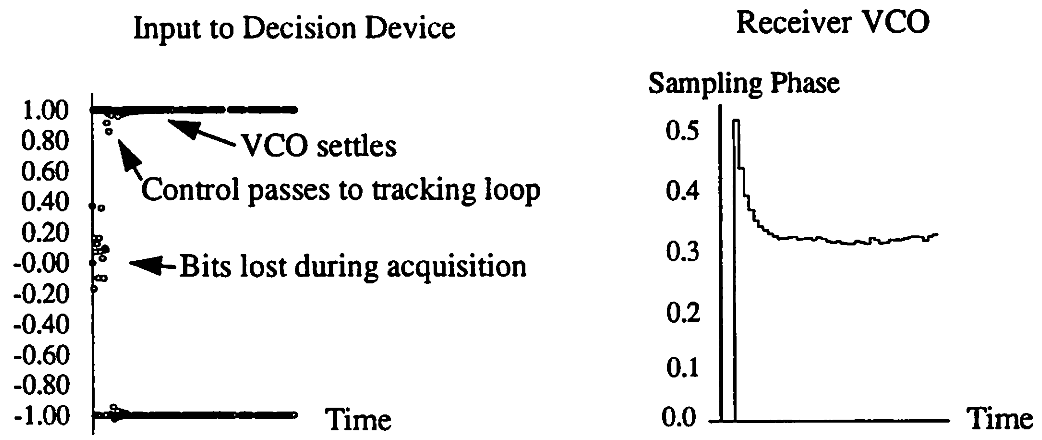

FIGURE 5.2 : Received data signal and receiver sampling phase shown as delay lock loop locks onto received signal

sampling phase settled to a (nearly) constant value. Output for a typical simulation is shown in Figure 5.2. Data output on the in-phase channel is plotted as a function of time. Bits are lost during acquisition mode operation since the proper code phase is unknown. Once control passes to the tracking loop, output quickly moves to the expected antipodal values. The sampling phase (relative to one chip time) is also illustrated in this figure. Acquisition mode places the phase within one half chip time of the optimal value. The DLL then quickly settles to a phase value near 0.35. Another simulation execution with multiple users and a multipath channel is shown in Figure 5.3. Here the effects of interference on the received signal are evident. The receiver sampling phase wanders more, yet still remains relatively stable. This is largely due to the fact that the power in the pilot signal is held fixed at 20% of the total transmit power in the system. Further simulation showed that the DLL functioned effectively even without a continuously variable sampling phase adjust. Monte Carlo simulation detected minimal performance degradation when the sampling phase was quantized to the nearest value in the set {0, 0.25, 0.5, 0.75}. The InfoPad custom chip is designed to allow this quarter phase adjust.

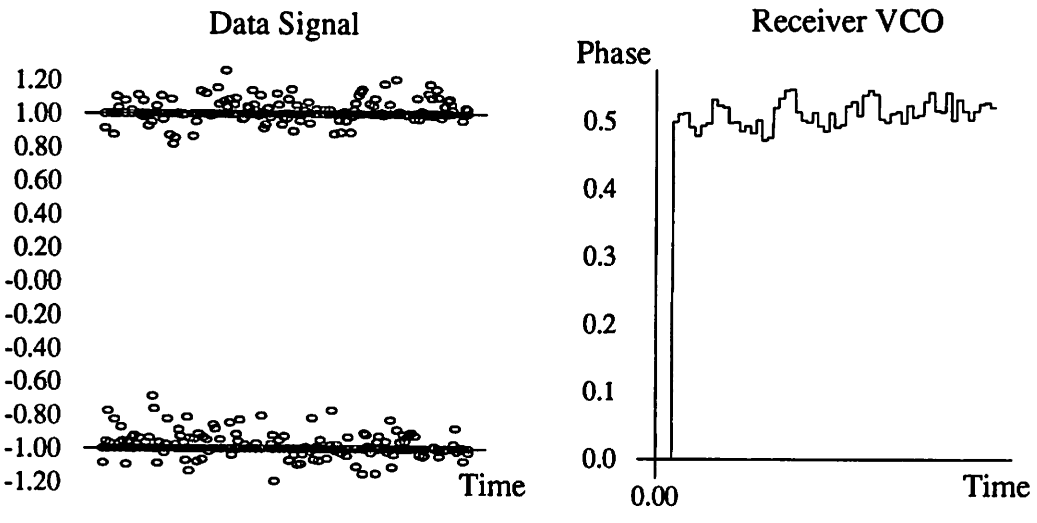

FIGURE 5.3 : Received data signal and receiver sampling phase shown for a multipath channel with multiple users.

5. 2. 2 Effectiveness of Spreading

The choice of a spreading factor of 64 was verified by measuring the total normalized integrated receive power in the band of interest at the front end of the receiver for a number of random channel realizations. For these channel models the mean normalized integrated receive power is a function of link distance only and falls off at a rate of (distance)$^{-2.6}$. Spreading the signal helps average out deep fades and reduce the variation of integrated receive power about the mean for different channels. This averaging helps relax the AGC requirements, making it very unlikely that the signal is in a deep fade across the entire bandwidth of interest. A plot is shown in Figure 5.4 where the dashed line represents the d$^{-2.6}$ expected mean, and the standard deviation of received power for a given link distance is approximately 3 dB.

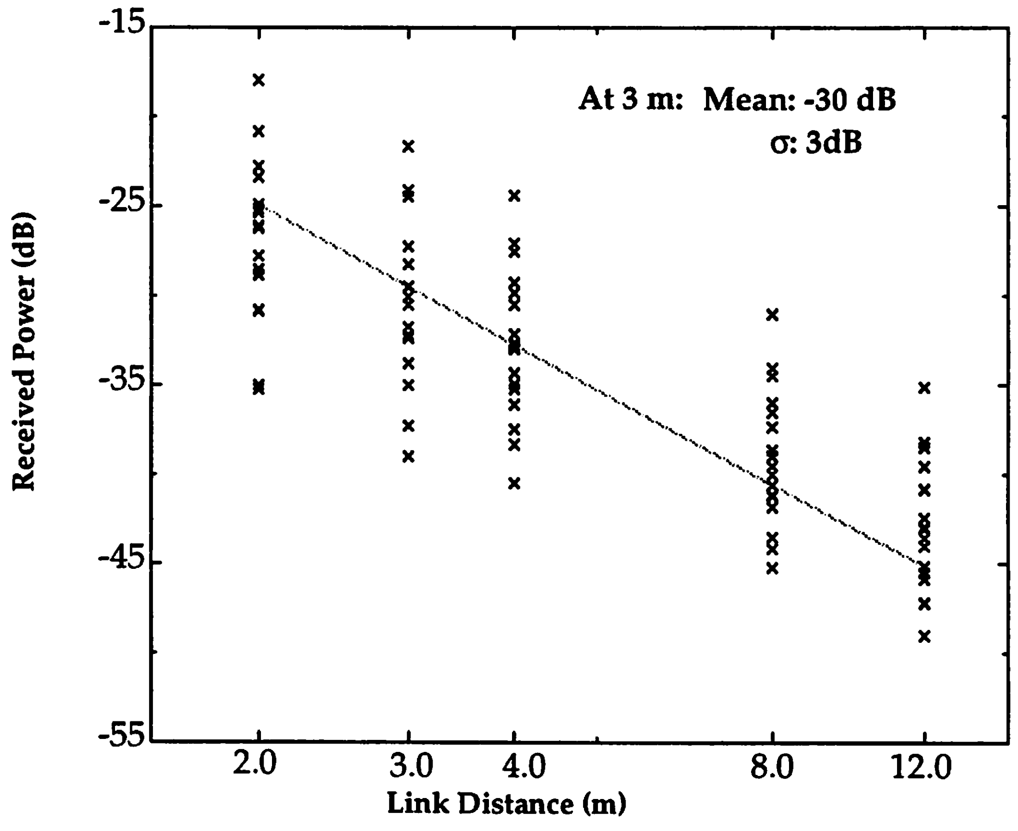

FIGURE 5.4 : Normalized integrated receive power for several simulated channels

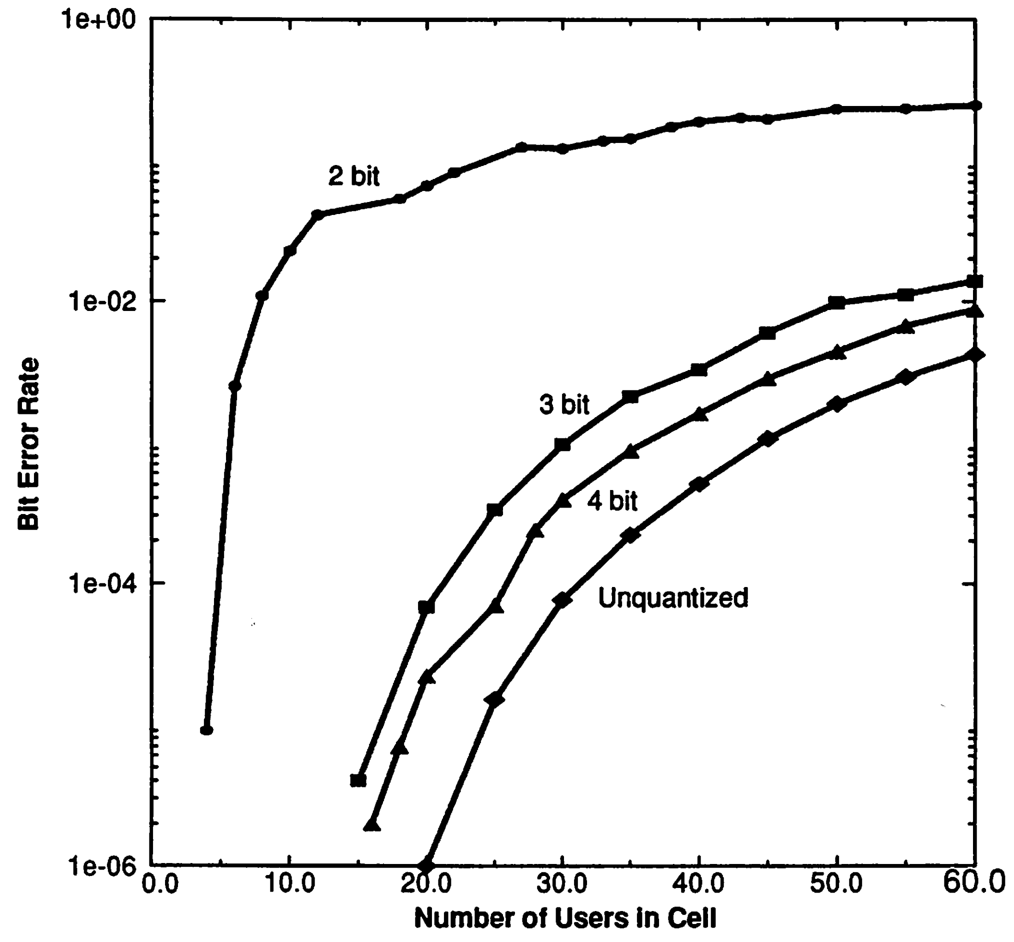

FIGURE 5.5 : Quantization effects on BER for a typical channel

5. 2. 3 Quantization Effects

Another hardware issue examined through simulation was the impact of quantizer resolution on bit error rate performance. The results for a typical channel are shown above in Figure 5.5 where the bit error rate as a function of the number of cell users is shown for different quantizer resolutions.

From the plot, it is apparent that there is little improvement in BER as the number of bits increases above four. This is intuitive. This signal arriving at the receiver consists of the desired user's signal, interference, and other system noise. Quantization contributes an additional noise term. A quantization noise magnitude substantially below interference and other noises has a negligible performance impact.

However, if the quantizer order is reduced sufficiently (below 3 bits), the quantization noise term will dominate.The attached plot illustrates that using fewer than three bits results in a significant BER penalty, while increasing the number of bits above four does little to improve performance. It should be noted that this plot implicitly assumes perfect AGC performance, allowing the quantizer to take full advantage of its dynamic range. Imperfect AGC operation generally results in a loss of dynamic range approximately equivalent to one-half bit reduction in quantizer performance. For this reason, the custom chip set relies on a four bit quantizer to ensure performance above the threshold.

5. 2. 4 BER Performance

5. 2. 4. 1 Simplifications

In discussing the bit error rate performance on the link, several qualifiers must be included. This simulation work does not attempt to provide average bit error rates over the link. Each simulation execution generates a fixed, statistical channel and uses the same channel throughout the length of the simulation run. As such, there is no provision for non-stationary effects, mobile terminal movement, etc. A more accurate estimate for the average BER would require inclusion of such factors. The most obvious extension would be to consider the "fixed" channel to be one time-varying state in a Markov process. Estimates of the transition probabilities from state to state would provide a means to compute an "average" bit error rate.

Another important simplification is the consideration of single base station transmission only; no provision is made for additional cells. Typically, base station multiplicity is modeled as additive white Gaussian noise at the receiver front end. Consequently, these results are easily extensible to multiple cell scenarios. For reasons explained previously, background and thermal noises are neglected as small by the interference-limited assumption. The following BER curves serve primarily to quantify and compare the impact of signal interference arising from multipath arrivals with theoretical values.

5.2.4.2 Predicted BER Curves

In chapter four, a Gaussian approximation for the channel-dependent multipath interference was proposed, and an expression for the probability of error was derived in Equation 4.30. The validity of this expression is to be examined here. For comparison purposes, predicted BER curves are shown below in Figure 5.6 for various values of the channel parameter α where

$$\alpha = \sum_{l=1}^{L-1} \left(\frac{\beta_l^2}{\beta_0^2} \right) \qquad \text{(EQ 5.1)}$$

and represents the ratio of total power in all interfering channel delay paths to power in the dominant (signal) path. It should be noted that the BER is a strong function of this

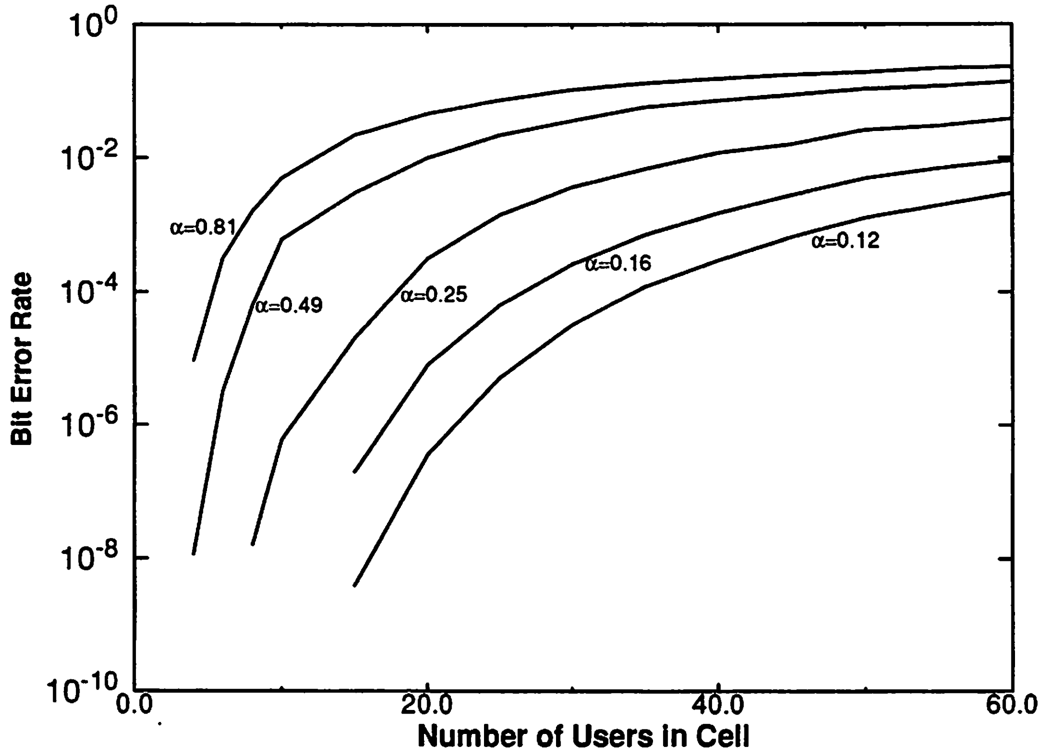

FIGURE 5.6 : Predicted BER curves illustrating channel-dependent interference as function of α.

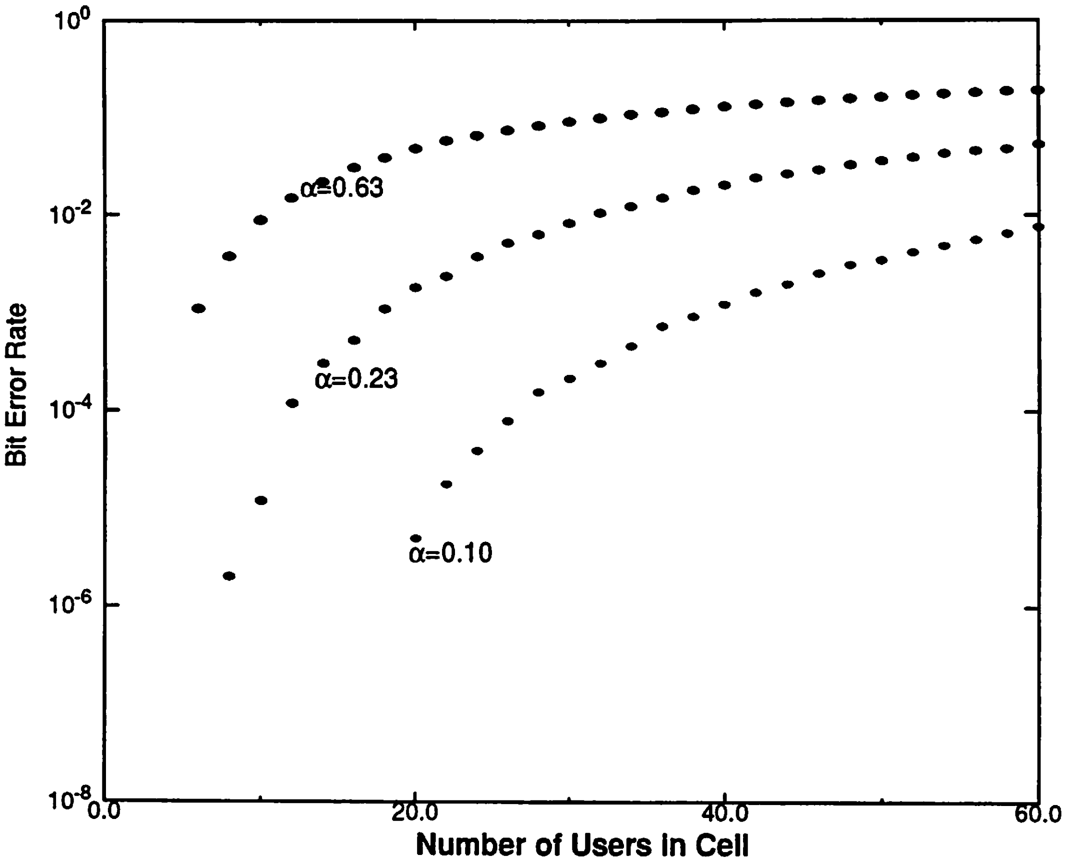

FIGURE 5.7 : Simulated BER curves illustrating channel-dependent interference as a function of α.

parameter. As expected, larger values of α correspond to increased interference levels and error rates.

5. 2. 4. 3 Simulated BER Curves

In Figure 5.8 three representative bit error rate curves are displayed. The same basic trend is observed for these channels as for the predicted BER curves. Once again, the bit error rate is observed to be a strong function of the cumulative energy in delayed paths.

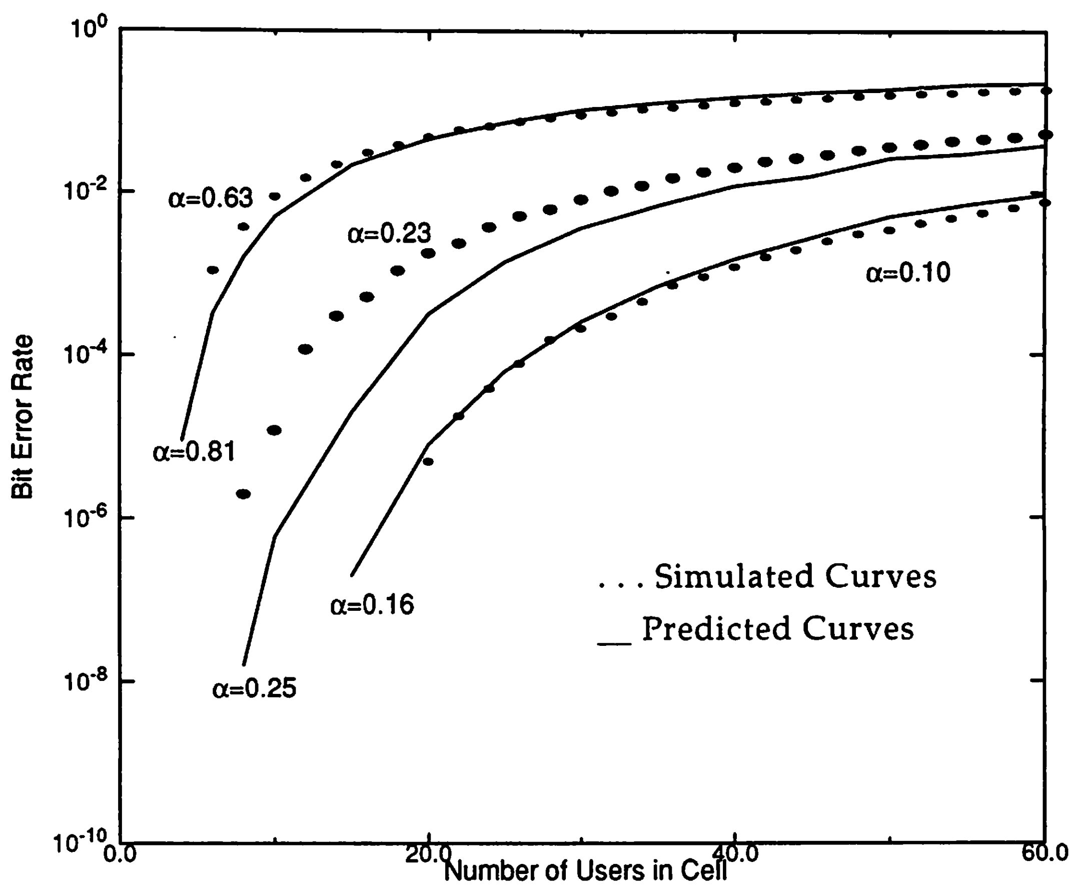

FIGURE 5.8 : Channel induced interference with no quantization (simulated data overlaid on predicted curves)

5. 2. 4. 4 Comparison of Simulated and Predicted Curves

Figure 5.8 shows the simulated curves overlaid on the predicted BER curves for comparison. Two observations should be noted. First and foremost, the general shape and form of the curves confirm the analysis performed previously. Secondly, simulated performance as a function of α appears to be slightly worse than predicted. This could be explained by the fact that the codes do, in fact, exhibit some correlations, so that additional noise may be added into the system. A more accurate estimate of the partial code correlations may yield a slightly larger value of α for these channels. In any case, the critical insight here is that a simple Gaussian interference approximation appears to provide a reasonable model for these channels.

5. 2. 5 RAKE Combining Results

The final issue examined in this research is the gains provided with the RAKE receiver. The theory developed earlier in this report indicated that the presence of signal-dependent interference required a rescaling of the optimal combining coefficients. In particular, it was shown that for a two tap channel with chip-spaced impulse response given by

$$h[k] = [1, ae^{j\theta}, 0, \ldots] \qquad \text{(EQ 5.2)}$$

the optimal combining coefficients are unity for the on-time finger and

$$c_1 = a^3 e^{-j\theta} \qquad \text{(EQ 5.3)}$$

for the one chip delayed finger. The implication of the analysis was that RAKE combining for the expected channel conditions will yield insignificant gains. An effort

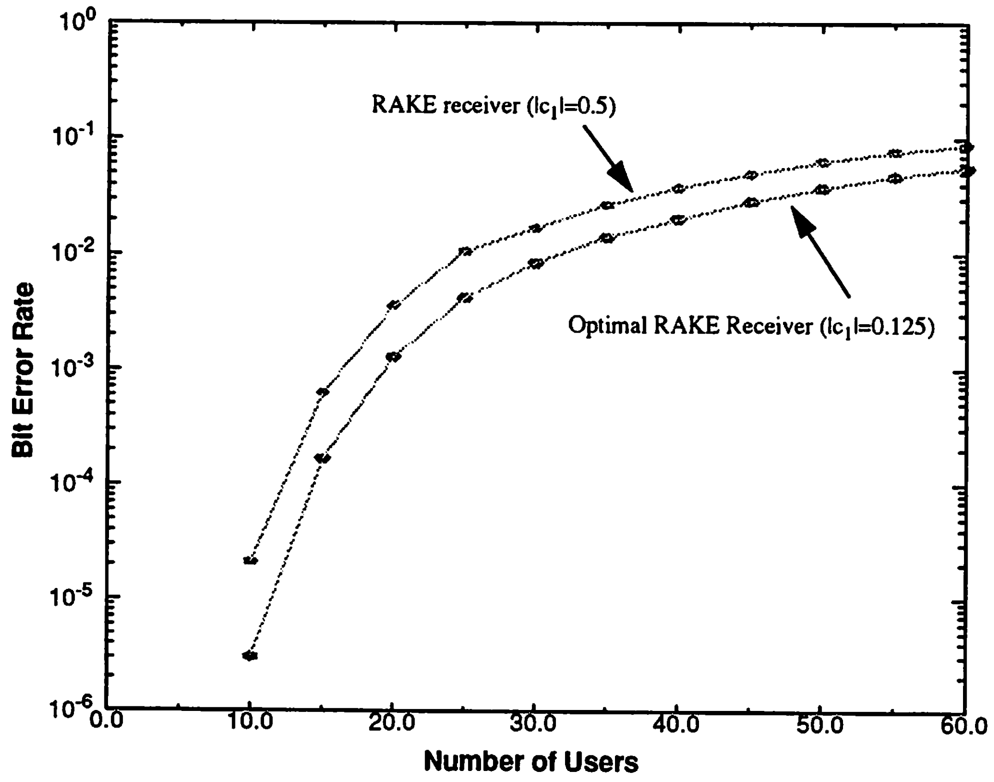

FIGURE 5.9 : Illustration of RAKE combining gains.

was made to test this hypothesis, and the results are shown in Figure 5.9. An artificial multipath channel whose impulse response is [1, $0.5e^{-j304}$, 0, ...] was used in conjunction with a two-finger modified RAKE receiver. Three different combining ratios were used, and the optimal (as expected) performance occurred when a cubic magnitude ratio for the second finger was used. Even then, the optimal ratio provides little improvement over a single finger. The obvious implication is that a RAKE combining receiver should not be employed in the InfoPad system. These results are similar, though slightly different from those reported in [Nore94]. There, equal gain combining on the RAKE fingers was employed, and the author notes that signal energy subsequently increases as anticipated. However, under certain scenarios the amount of additional noise added by this configuration can more than offset the signal increase, resulting in performance degradation as additional fingers are added. The results of this study confirm that author's observations.

6 Conclusion

This report documents research into three areas of the InfoPad project: mobile receiver synchronization, multipath equalization and bit error rate performance on the link. Analysis, combined with a sophisticated system-level simulation were employed to assist in making certain hardware design choices related to the downlink radio system design.

Simulation verified that an exhaustive search algorithm and a simple delay locked loop would provide adequate timing recovery fot the mobile receiver. Additionally, simulation suggested that a four bit A/D converter and a quarter chip sampling phase resolution would provide acceptable performance, thus simplifying the hardware design.

The use of a RAKE receiver to combat multipath signal degradation was investigated, and shown (through analysis and simulation) to exhibit negligible performance gains. The RAKE receiver structure was originally developed for reception of a single data signal in the presence of multipath reflections. Efforts to employ the technique in a multi-user scenario with multipath reflections appear to be ineffective at best.

Simulated performance for the InfoPad system also indicates that bit error rates on the link are highly dependent on the amplitude of the reflection coefficients.

Analytic expressions for this dependence were derived and compared to simulated performance. Results indicate that with a spreading factor of 64, it may be difficult to achieve adequate bit error rate performance on the link for 50 users operating at masimum bit rates with the existing channel models.

Future work will focus on techniques to reduce this multiple access interference resulting from multipath arrivals. In particular, equalization and interference cancellation techniques will be investigated in an effort to determine whether they can provide adequate performance gains. Preliminary results indicate that pilot channel estimation techniques previously employed in the RAKE structure, combined with simple equalization approximations, may yield dramatic performance improvements. In some sense, the RAKE receiver does not exploit the fact that the correlation structure of the interference is known. As long as a reasonable estimate of the channel is provided, interference at the receiver is no longer a random variable, but a known quantity. It is hoped that this knowledge may provide significant performance improvements.

References

[Beau75] K. G. Beauchamp, *Walsh Functions and Their Applications*. London: Academic Press, 1975.

[Bree91] M.A. Bree, J.E. Salt, D.E. Dodds, "A PN Code Acquisition Method Using Amplitude Modulated Pilot Tones", publication of WIRELESS '91 Conference?.

[Camag93] J. Camagna, "An Analysis of the InfoPad Downlink," Masters Thesis, UC Berkeley, May 1993.

[Cout94] M.A. Couture, J.P.M.G. Linnartz, "Improved Channel Modeling for Performance Analysis of a DS-CDMA Link with Exponential Delay Profile," PIMRC '94, May 1994.

[Davies73] N.G. Davies, "Performance and Synchronization Considerations," AGARD Lecture Series on Spread Spectrum Communications, July 1973.

[Gil91] K. Gilhousen, I. Jacobs, R. Padovani, A. Viterbi, L. Weaver, C. Wheatley, "On the Capacity of a Cellular CDMA System," *IEEE Trans. on Vehicular Tech.*, vol. 40, no. 2, pp. 310-312, May 1991.

[Grob90] U. Grob, A. Welti, E. Zollinger, R. Kung, H. Kaufmann, "Microcellular Direct-Sequence Spread-Spectrum Radio System Using N-Path System RAKE Receiver," *IEEE Journal of Selected Areas in Communications*, vol. 8, no. 5, June 1990

[Harris73] R.L. Harris, "Introduction To Spread Spectrum Techniques," AGARD Lecture Series on Spread Spectrum Communications, July 1973.

[Holmes82] J. Holmes, *Coherent Spread Spectrum Systems*. New York: Wiley, 1982.

[Hurst92] P. Hurst, Personal Report of Research Activities, December 1992.

[Lee91] W.C. Y. Lee, "Overview of Cellular CDMA, *IEEE Trans. on Vehicular Tech.*, vol. 40, no. 2, May1991.

[Nore94] D.L. Noreaker, M.B. Pursley, "On the Chip Rate of CDMA Systems with Doubly Selective Fading and Rake Reception," *IEEE Journal of Selected Areas in Communications*, vol. 12, no. 5, June 1994.

[Pick82] R.L. Pickholtz, D.L Schilling, and L.B. Milstein, "Theory of spread-spectrum communications -- A tutorial," *IEEE Trans. Commun.*,vol. COM-30, pp. 855-884, May 1982.

[Pick91] R. L. Pickholtz, L.B. Milstein, and D.L. Schilling, "Spread Spectrum for Mobile Communications," *IEEE Trans. on Vehicular Tech.*, vol. 40, no. 2, pp. 313-317, May 1991.

[Proak89] J.G. Proakis, *Digital Communications*. New York: McGraw-Hill, 1989, ch. 7, 8.

[Qual92] "An Overview of the Application of Code Division Multiple Access (CDMA) to Digital Cellular Systems and Personal Cellular Networks," published by QUALCOMM Inc., 1992.

[Saleh87] A.M. Saleh, R.A. Valenzuela, "A Statistical Model for Indoor Multipath Propagation," IEEE Journal of Selected Areas in Communications, vol. SAC-5, no. 2, pp 128-137, Feb. 1987.

[Sheng91] S.. Sheng, "Wideband Digital Portable Communications: A System Design", Masters Thesis, UC Berkeley, December 1991.

[Subr91] R. Subramanian, "Synchronization Systems for Spread-Spectrum Receivers," Ph.D. Thesis, UC Berkeley, November 1991.

[Yost82] R.A. Yost, R.W. Boyd, "A Modified PN Code Tracking Loop: Its Performance Analysis and Comparitive Evaluation," *IEEE Trans. Commun.*, vol. COM-30, pp. 1027-1036, May 1982.

Appendix I

This project neglects most multiple base station issues. Extending the results of this paper, however, an estimate of the bit error rate on the link when multiple base stations are operating may be derived. A simple approximation for the bit error rate on the downlink is given by

$$P_e = erfc(\sqrt{\Upsilon}) \cdot \left[1 - \frac{1}{4} erfc(\sqrt{\Upsilon})\right]$$

where

$$\Upsilon = \frac{E_b}{N} = \frac{Z_k E\left[\beta_{max}^2\right]}{\frac{1}{M} \cdot \left(Z_k K \alpha + \sum_{n = 1, n \neq k}^{N_t} Z_n\right)}$$

$$Z_n = P_n \cdot d_n^{-\lambda_n}$$

$$\alpha = \sum_{l=0,\, l \neq l_{max}}^{L-1} E\left[\beta_l^2\right]$$

and the following definititions are used:

The mobile is assumed to be in cell k
E_b is the received signal energy per bit
N_t is the number of base station transmitters
P_n is the total transmit power of base station n
Z_n is the (scaled) interference power at the mobile unit due to transmission from base station n
d_n is the distance from the mobile to base station transmitter n
λ_n is the roll-off factor for a signal traveling from base station n to the mobile
M is the spreading factor in the system
K is the number of users active in the mobile unit's cell
β_l is the magnitude of the lth tap in a chip-spaced channel impulse response for the mobile unit's cell
β_{max} is the magnitude of the largest component in the chip-time spaced impulse response for the mobile unit's cell

Appendix II

Ptolemy Code

```
defstar {
        name { AcqTrack9 }
        domain { SDF }
        author { Craig Teuscher (4/93)}
        version { @(#)AcqTrack9.pl     1.6 1/30/93 }
        copyright {
Copyright (c) 1990, 1991, 1992 The Regents of the University of California.
All rights reserved.
See the file ~ptolemy/copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
        }
        location { user pallette }
        desc {
         This star performs acquisition and tracking of a complex valued spread
         spectrum signal. Data is assumed to arrive with two samples per chip
         period, the on-time and the late. From the previous chip's late sample,
         the next chip's early sample can be determined. PN sequences on
         the real and imaginary channels are generated, and correlation
         with the incoming data is performed. If correlation falls below
         the user-specified threshold, an acquisition search is initiated which
         performs a serial search on the incoming data stream to determine the PN
         code phase offset which maximizes the correlator. Once this maximum is
         found, the code phase is adjusted appropriately, and the star resumes tracking
         mode operation using a DLL.
            Outputs include the PN phase (for the Walsh correlators to use), the values
         output which shows the threshold value of the correlator, a VCO sampling
         phase signal that  is fed back to the sampler, and a coeffs output which
         provides the on-time, one and two chip late correlator outputs of the pilot signa
         This channel estimate information is used to construct the equalizer filter.
         Several user specified parameters are allowed. These should be self-explanatory.
        }

        input { name { data } type { complex } }
        input { name { data_late } type { complex } }

        output { name { PNphase } type { int } }
        output { name { values } type {float} }
        output { name { phase } type { float } }
        output { name { coeffs } type { complex } }

        defstate {
                name { sequence_length_in_chips }
                type { int }
                default { sequence_length_in_chips }
                desc { PN sequence length in chips }
        }

        defstate {
                name { chips_to_integrate }
                type { int }
                default { chips_to_integrate }
                desc { number of chips to integrate over in correlation }
        }
        defstate {
                name { mlsrLength }
                type { int }
                default { mlsrLength }
                desc { Length of PN scrambling sequence }
        }

        defstate {
                name { spreadCode }
                type { int }
                default { spreadCode }
                desc { Spreading code }
        }
        defstate {
                name { gain }
                type { float }
                default { gain }
                desc {  Gain on DLL }
        }
        defstate {
                name { threshhold }
                type { float }
                default { threshhold }
                desc {  Threshhold to determine lock }
        }

        ccinclude { <math.h> }
        protected {
                    int acqcounter,PNcounter,Corrcounter,full_corr_counter,acquire,N;
                    int spread[5000];
                    float samplephase;
                    Complex earlysum,latesum,onechiplatesum,twochiplatesum,ontimesum;
                    Complex ontimecoeff,onechiplatecoeff,twochiplatecoeff;
                    Complex dataarr1[5000],dataarr2[5000];
        }

        start {
                N=int(sequence_length_in_chips);
                acqcounter=N*2;
                PNcounter=Corrcounter=full_corr_counter=acquire=0;
                samplephase=0;
                earlysum=latesum=onechiplatesum=twochiplatesum=ontimesum=Complex(0.0);
                ontimecoeff=onechiplatecoeff=twochiplatecoeff=Complex(0.0);
                loadPNseq(spread);
                data_late.setSDFParams(1,1);
                coeffs.setSDFParams(3,2);
        }

        go {
                #define FULL_CORRELATIONS 1
                float early,late,normalize_factor;

           // Output the PN phase

                PNphase%0 << PNcounter;

           // If in acquisition mode, decrement the counter
           // and store the two incoming samples if they will
           // be used in the correlation (depends on the number
           // of chips that will be integrated over).

                if (acquire) {
                   acqcounter--;
                   if (acqcounter>=(N-int(chips_to_integrate))) {
                        dataarr1[acqcounter]=Complex(data%0);
                        dataarr2[acqcounter]=Complex(data_late%0);
                   }

           // If the acquisition counter has reached zero, correlate with the
           // two arrays to find the optimal PN code phase offset, and adjust
           // the samplephase so that the ontime or early stream is used as the
           // data stream, depending on which array gives the larger correlation
           // peak.  Resets all of the counters and correlator outputs.

                   if (acqcounter==0) {
                    correlate(dataarr1,dataarr2,&samplephase,&PNcounter);
                    acqcounter=2*N;
                    acquire=Corrcounter=full_corr_counter=0;
                    earlysum=latesum=onechiplatesum=twochiplatesum=ontimesum=Complex(0
                }
            }

        // If not in acquisition mode, accumulate in the early, late,
        // ontime, onechiplate and twochiplate correlators, and increment
        // the counters.

            else {
                earlysum+=Complex(data_late%1)*float(spread[PNcounter]);
                latesum+=Complex(data_late%0)*float(spread[PNcounter]);

        // Make sure that the PN code array wraps around properly at the end.

                if ((PNcounter==0) || (PNcounter==1)) {
                    if (PNcounter==0) {
                        onechiplatesum+=Complex(data%0)*float(spread[N-1]);
                        twochiplatesum+=Complex(data%0)*float(spread[N-2]);
                    }
                    if (PNcounter==1) {
                        onechiplatesum+=Complex(data%0)*float(spread[0]);
                        twochiplatesum+=Complex(data%0)*float(spread[N-1]);
                    }
                }
                else {
                    onechiplatesum+=Complex(data%0)*float(spread[PNcounter-1]);
                    twochiplatesum+=Complex(data%0)*float(spread[PNcounter-2]);
                }
                ontimesum+=Complex(data%0)*float(spread[PNcounter]);
                Corrcounter++;
                PNcounter++;
            }

        // If the PN phase counter is at its maximum value, check to see if a full
        // correlation has been performed (removes transient effects which result
        // when an acquisition is performed and correlation occurs only over a few
        // samples).  The early and late correlation values are then normalized for
        // convenience.

            if (PNcounter==N) {
               if (Corrcounter==N) {
                  normalize_factor=float(N)*1.414;
                  early=abs(earlysum)/normalize_factor;
                  late=abs(latesum)/normalize_factor;
                  full_corr_counter++;

        // If energy in the early and late falls below threshold, set flag
        // to reacquire; otherwise, DLL adjusts the sampling phase. Store the
        // ontime, onechiplate, and twochiplate correlator coefficients.

                  if ((early+late) < float(threshhold)) acquire=1;
                  else samplephase += (early-late)*float(gain);
                  if (full_corr_counter==FULL_CORRELATIONS) {
                     ontimecoeff=ontimesum/(FULL_CORRELATIONS*normalize_factor);
                     onechiplatecoeff=onechiplatesum/(FULL_CORRELATIONS*normalize_fact
                     twochiplatecoeff=twochiplatesum/(FULL_CORRELATIONS*normalize_fact
);
                     full_corr_counter=0;
                     }
               }
        // Output the threshold value on the values stream, then reset variables.

                values%0 << (early+late);
                PNcounter=Corrcounter=0;
                early=late=0;
                earlysum=latesum=onechiplatesum=twochiplatesum=ontimesum=Complex(0.0);
              }

        // Otherwise output a zero on the values stream.

              else values%0 << 0;

        // Keep the samplephase bounded between -0.5 and 1.1, and output the
        // correlator coefficients

              if (samplephase>1.1) samplephase=1.1;
              if (samplephase<-0.5) samplephase=-0.5;

              phase%0 << samplephase;

        // Forget about the twochiplate correlator output.  The energy is low enough th
        // it becomes insignificant in most cases.
          //   coeffs%2 << 0.0;
              coeffs%2 << twochiplatecoeff;
              coeffs%1 << onechiplatecoeff;
              coeffs%0 << ontimecoeff;

      } // end go

      method {
           name { loadPNseq }
           access { protected }
           arglist { "(int *spread)" }
           type { void }

        // Generates the PN sequence used for descrambling.

           code {
               int i,k,temp,CodeLength;
               int lin[14],regstate[14];
               CodeLength = 1;
               for (i=0; i < int(mlsrLength); i++) {
                   lin[i] = (int(spreadCode)/CodeLength) % 2;
                   regstate[i]=0;              // initialize regstate
                   CodeLength *= 2;          // increment spreadFactor
               }
               regstate[int(mlsrLength)-1] = 1;


               for (i=0;i<(CodeLength-1);i++) {
                   spread[i] = (regstate[0]==1) ? 1: -1;

                   temp=0;

                   for (k=0; k<=(int(mlsrLength)-1); k++) {
                       temp += lin[k]*regstate[k];
                   }
                   for (k=(int(mlsrLength)-2); k>=0; k--) {
                       regstate[k+1] = regstate[k];      // left shift
                   }
                   regstate[0] = temp % 2;
               }
               spread[CodeLength-1]=1;       // extra bit added
```

```
            } // end code
        } // end method

        method {
            name { correlate }
            access { protected }
            arglist { "(Complex *data, Complex *datalate,float *samplephase,
                        int *PNcounter)" }
            type { void }

            code {
               float Cmax, Cofn1,Cofn2;
               Complex ctop1,ctop2;
               int N,n,m,nmax,maxcorr;

               Cmax=0;
               nmax=maxcorr=0;
               N=int(sequence_length_in_chips);


            // At each code phase offset, compute the correlator value by
            // integrating over an interval of length chips_to_integrate
            // Do this for the ontime and late streams so that this search
            // is performed with a resolution of one half a chip interval.
            // Take the magnitude of this complex correlator and select the
            // largest.

               for ( n = 0; n < N; n++ ) {
                      ctop1=ctop2= Complex(0.0);
                      for ( m = 0; m < int(chips_to_integrate); m++ ) {
                          ctop1 += data[2*N - 1 - ( n + m )] *
                                      float(spread[m]);
                          ctop2 += datalate[2*N - 1 - ( n + m )] *
                                      float(spread[m]);
                      } // end for
                      Cofn1 = abs(ctop1);
                      Cofn2 = abs(ctop2);

            // If the ontime or late correlation yields the largest value
            // encountered thus far, make it the maximum and store the index.

                      if ((Cofn1>Cmax) || (Cofn2>Cmax)) {
                          if (Cofn1 > Cofn2) {
                             Cmax = Cofn1;
                             nmax = n;
                             maxcorr=1;
                          } // end if
                          else {
                             Cmax = Cofn2;
                             nmax = n;
                             maxcorr=2;
                          } // end if
                      }
               } // end for

            // If the late stream yields the maximum value of all possible
            // offsets, then increment the phase (or decrement it) by 0.5
            // so that the late stream will become the data stream. Set the PN
            // counter to its proper value, given the offset which makes the
            // correlator output maximum.

               if (maxcorr==2) {
                  if (*samplephase<0.5) *samplephase += 0.5;
                  else *samplephase -= 0.5;
```

```
                }
                *PNcounter=N-nmax;
             } // end code
       } // end method

} // end defstar
```

```
defstar {
        name { BER2 }
        domain { SDF }
        desc {
           This star functions as a simple bit error rate tester.  The 'original'
input is the original data generated.  The 'data' input represents the received
data signal, assumed to be hard-limited to the same levels as the original data.
A delay is specified between the original and received data streams, requiring
the user to indicate the latency associated with the received data.  Also, a training
length may be specified during which no BER measurement is made.  A simple
equality comparison is made between the two streams, and two outputs are
generated.  The 'errors' output keeps a running total of the number of errors
which have occurred since the simulation commenced.  The 'rate' output divides
this quantity by the number of bits which have been processed, thus providing
a BER for the data streams..

        }
        version {@(#)SDFBER2.pl 2.12 11/25/92}
        author { Craig Teuscher 2/93 }
        copyright {
Copyright (c) 1990, 1991, 1992 The Regents of the University of California.
All rights reserved.
See the file ~ptolemy/copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
        }
        location { user.pal }
        explanation {
        }
        input {
                name { original }
                type { float }
        }
        input {
                name { data }
                type { float }
        }
        output {
                name { errors }
                type { float }
        }
        output {
                name { rate }
                type { float }
        }

        defstate {
                name { training_length }
                type { int }
                default { training_length }
                desc { The number of bits lost during the training sequence.}
        }

        defstate {
                name { delay }
                type { int }
                default { delay }
                desc { The number of delays between 'original' and 'data'.}
        }
        protected {
               int numberofbits,numberoffirings;
               int totalerrors;
        }
        start {
               numberofbits=0;
```

```
                numberoffirings=0;
                totalerrors=0;

                original.setSDFParams(1, int(delay));
        }
        go {

        // Check to see if enough firings have taken place to pass the training
        // sequence bits and to allow for the delay between the original stream
        // and the data stream.

            if (numberoffirings >= int(training_length)+int(delay)) {

        // increment the bit counter and the error counter if  appropriate.
                  numberofbits++;
                  if (float(original%int(delay)) != float(data%0)) totalerrors++;
                  errors%0 << float(totalerrors);
                  rate%0 << float( float(totalerrors)/float(numberofbits));
            }

        // If we are in the training sequence, output zeros and increment the
        // counters.

          else  {
                 numberoffirings++;
                 errors%0 << 0.0;
                 rate%0 << 0.0;
            }
        }
}
```

```
defstar {
        name { CWalshCorr }
        domain { SDF }
        desc {
            This star functions as a Walsh sequence correlator.  The user parameter
        determines which Walsh code will be used to correlate the incoming waveform.
        The spreadfactor parameter determines the decimation rate.  (Spreadfactor)
        tokens are consumed and a single output is generated on each firing. The incoming
        complex data stream is multiplied by the desired Walsh sequence, and the result
        is integrated over the length of the Walsh sequence.  When the end of the sequence
        is reached, the final value is "dumped" and the correlator is reset to begin anoth
        correlation.  The PNphase input is a "clock" which ensures that the Walsh correlat
        and the PN generator remain synchronized.
        }
        version {@(#)SDFWalshCorr.pl    2.12 11/25/92}
        author { Craig Teuscher 2/93 }
        copyright {
Copyright (c) 1990, 1991, 1992 The Regents of the University of California.
All rights reserved.
See the file ~ptolemy/copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
        }
        location { user.pal }
        explanation {
        }

        hinclude {
            <math.h>
        }
        input {
                name { data }
                type { complex }
        }
        input {
                name { PNphase }
                type { int }
        }
        output {
                name { output }
                type { complex }
        }
        defstate {
                name { user }
                type { int }
                default {"0"}
                desc { The user number.}
        }
        defstate {
                name { spreadfactor}
                type { int }
                default {"64"}
                desc { The spreading factor. }
        }
        protected {
               int walsh[128];
               Complex correlation;
        }
        start {
               int i,r,bits,exponent;
               int userbits[8], timebits[8];

               correlation=Complex(0.0,0.0);
               bits =  (int) (log10(double(spreadfactor))/log10(2.0));
       // Generate the Walsh codes through the following procedure:
        // Convert user to binary representation stored in userbits
                convertobits(user,bits,userbits);
                userbits[bits]=0;

        // For each chip interval convert the time index i to a binary
        // represenation stored in timebits.

                for (i=0;i< int(spreadfactor);i++) {
                    exponent=0;
                    converttobits(i,bits,timebits);
                    timebits[bits]=0;

        // Given the userbits and timebits, apply the formula for the Walsh
        // sequence value. Make the output binary antipodal.

                    for (r=0;r<bits;r++) {
                        exponent += userbits[bits-1-r] * (timebits[r] + timebits[r+1]);
                    }
                    walsh[i] = (exponent%2==0) ? 1: -1;

                }
                PNphase.setSDFParams(int(spreadfactor),int(spreadfactor)-1);
                data.setSDFParams(int(spreadfactor),int(spreadfactor)-1);
         }
         go {
             int i,phase,index;

         // Determine phase of the Walsh clock related to the phase of the PN clock

             phase = int(PNphase%(int(spreadfactor)-1)) % int(spreadfactor);

         // Compute the running correlation sum
             for (i=0;i<int(spreadfactor);i++) {
                 index =  (i+phase) % int(spreadfactor);
                 correlation += walsh[index]* Complex(data%(int(spreadfactor)-1-i));

         // When the end is reached, dump output and reset correlator to
         // begin a new correlation.

                 if (index==(int(spreadfactor)-1)) {
                     output%0 << Complex(correlation * float(1/float(spreadfactor)) );
                     correlation= Complex(0,0);
                 }
             }
         }
         method {
               name { converttobits }
               access { protected }
               arglist {"(int number, int numberofbits, int *bits)" }
               type { void }
               code {
                    int i,divisor;

         // Converts a decimal number into its binary representation

                    divisor=1;

                    for (i=0;i<numberofbits;i++) {
                        bits[i]= (number/divisor) %2;
                        divisor*=2;
                    }
               }
         }
```

```
}
```

```
defstar {
    name { CompSamp5 }
    domain { SDF }
    author { Craig Teuscher }

    location { User Pallette }

    desc {
This star acts as a sampler/decimator to provide the desired samples
requested by the sample_phase variable. The sample_phase input variable,
between 0 and 1, requests that the output sample be taken at the time
T_chip*sample_phase. Two samples are generated, the on-time and the late.
    }

    input {
        name { in }
        type { complex }
    }

    input {
        name { sample_phase }
        type { float }
        // 0 <= sample_phase <= 1 -> gives sample position in the chip
        // interval
    }

    output {
        name { output }
        type { complex }
        // sample value at ideal chip sampling time
    }

    output {
        name { out_late }
        type { complex }
        // sample value 1/2 chip period after ideal chip sampling time
    }

    defstate {
        name { samples_per_chip }
        type { int }
        default { "samples_per_chip" }
        desc { Number of input samples per chip period. }
    }


    hinclude {
        <iostream.h>
    }

    start {
        in.setSDFParams(int(samples_per_chip),3*int(samples_per_chip));
    }
    go {
        // WARNING - DANGER -- Assumes 0 freq offset!!
        // Sample values are being computed with
        // one clock cycle delay, which allows us to handle
        // sample_phase = 1 and sample_phase = 0.

        float interp_factor;
        float exact_interp_time;
        int left_sample_time;
        Complex ontime,late;
        if(float(sample_phase%0) < -0.5 || float(sample_phase%0) > 1.5) {
                Error::abortRun(*this, "Smpl_ph_det:: ERROR: need sample_phase abetween -0
                exit(1);
        }


     // should check for bad sample_phase (not between 0 and 1?)


     // compute ideal sample value

        exact_interp_time = 2.0 * float(samples_per_chip) -
                            (float(samples_per_chip) * float(sample_phase%0)) -1;
        left_sample_time = (int) exact_interp_time +1;
        interp_factor = left_sample_time - exact_interp_time ;
     // interp_factor is used for a linear interpolation

        ontime = ( Complex(in%(left_sample_time)) * (1-interp_factor) +
                   Complex(in%(left_sample_time-1)) * interp_factor );

        ontime=Complex(ontime.real(),ontime.imag());

     // compute "late" sample
        exact_interp_time = 2.0 * float(samples_per_chip) - float(samples_per_chip)/2.0 -
                            (float(samples_per_chip) * float(sample_phase%0)) -1;
        left_sample_time = (int) exact_interp_time +1;
        interp_factor = left_sample_time - exact_interp_time ;


        late = ( Complex(in%(left_sample_time)) * (1-interp_factor) +
                 Complex(in%(left_sample_time-1)) * interp_factor );

        late=Complex(late.real(),late.imag());

        output%0 << ontime;
        out_late%0 << late;
    }
}
```

```
defstar {
    name { PN2 }
    domain { SDF }
    author { Culprit: Sam , modified by Craig Teuscher}

    location { RF Simulation library }

    desc {
Takes an input sequence and multiplies by an  MLSR descrambling sequence.
MLSR sequences are (2^length - 1) long.  An extra 1 is added on
to the end of the sequence to make the sequence (2^length) long. The
multiply consists of a scalar multiply of the complex data by the PN
descrambling sequence, and data is output at the same rate as it arrives.
  A phase input is also provided for synchronization purposes at the receiver.
The phase input indicates the phase of the PN sequence.
    }

    explanation {
    }


    input {
        name { data }
        type { complex }
    }

    input {
        name { phase }
        type { int }
    }

    output {
        name { signalOut }
        type { complex }
    }

    defstate {
        name { spreadCode }
        type { int }
        default { "spreadCode" }
        desc { Spreading code (MLSR polynomial coefficients); specified as an
               equivalent integer}
    }

    defstate {
        name { mlsrLength }
        type { int }
        default { "mlsrLength" }
        desc { Length of the MLSR.  The spreading factor is 2^mlsrLength - 1. }
    }



    protected {
        int spread[2048];
    }


    start {
        int i,k,temp,CodeLength;
        int lin[14],regstate[14];

    // This code generates the PN sequence and stores it in the spread array.
    // A single bit is appended to the code to make the length 2^N instead of
    // the traditional (2^N - 1).

        CodeLength = 1;
        for (i=0; i < int(mlsrLength); i++) {
            lin[i] = (int(spreadCode)/CodeLength) % 2;
            regstate[i]=0;              // initialize regstate
            CodeLength *= 2;            // increment CodeLength
        }
        regstate[int(mlsrLength)-1] = 1;

        for (i=0;i<(CodeLength-1);i++) {
            spread[i] = (regstate[0]==1) ? 1: -1;
            temp=0;
            for (k=0; k<=(int(mlsrLength)-1); k++) {
                temp += lin[k]*regstate[k];
            }
            for (k=(int(mlsrLength)-2); k>=0; k--) {
                regstate[k+1] = regstate[k];      // left shift
            }
            regstate[0] = temp % 2;
        }
        spread[CodeLength-1]=1;        // extra bit added
        }

    go {
        signalOut%0 << Complex( (float) spread[int(phase%0)] * Complex(data%0));
       }
}
```

```
defstar {
        name { PowerAGC }
        domain { SDF }
        desc {  This star models an AGC.  The signal arrives as a complex valued data
        stream.  The norm of the signal is averaged over the number of data points
        specified by the updategain parameter.  The gain is then updated (adjusted) so
        that the "power" of the signal has an average value equal to the specified
        powerlevel parameter.  The accumulator is reset and the process repeats.
        The output signal -- dataout -- consists of the input signal multiplied by the
        gain element.
        }
        version {@(#)SDFPowerAGC.pl     2.12 11/25/92}
        author { Craig Teuscher 2/93 }
        copyright {
Copyright (c) 1990, 1991, 1992 The Regents of the University of California.
All rights reserved.
See the file ~ptolemy/copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
        }
        location { user.pal }
        explanation {
        }

        hinclude {
            <math.h>
        }
        input {
                name { data }
                type { complex }
        }
        output {
                name { dataout }
                type { complex }
        }
        output {
                name { gainout }
                type { float }
        }
        defstate {
                name { powerlevel }
                type { float }
                default { powerlevel }
                desc { The normalized powerlevel desired.}
        }
        defstate {
                name { updategain }
                type { int }
                default { updategain }
                desc { The number of samples to average before updating gain.}
        }
        protected {
              int counter;
              float runningsum,gain;
        }
        start {
              runningsum=0;
              counter=0;
              gain=1.0;
        }
        go {

      // Accumulate the norm of the signal (after it is multiplied by the present
      // gain value). Increment the counter to keep track of the data points that
      // have been processed.
```

```
            runningsum += norm(gain*Complex(data%0));
            counter++;

      // The gain value is output for convenience, while the dataout stream consists
      // of the input stream multiplied by the gain value.

            gainout%0 << gain;
            dataout%0 << Complex(gain*Complex(data%0));

      // If enough data points have been processed, update the gain so that the average
      // powerlevel corresponds to the desired value.  Then reset the counter and accumula

            if (counter==updategain) {
               gain *= sqrt((float(powerlevel)*counter)/runningsum);
               counter=0;
               runningsum=0;
            }
        }

}// end defstar
```

```
defstar {
        name { WalshInt2 }
        domain { SDF }
        desc {
            This star functions as a Walsh interference generator and/or data
spreader. The spreadfactor determines the length of the Walsh sequence and should
be of length (2^n) where n is an integer.  Each user is assigned a unique
Walsh sequence, and data bits are mapped into the appropriate Walsh sequence
by multiplying the sequence by +/- 1 depending on whether the information bit
is +/- 1. The 'users' parameter allows one to specify all of the users in
the system in one star. Simply list the user numbers (ranging from 0 to (spreadfactor-1))
where 0 represents the pilot tone (DC) channel.  For a single user, the incoming data stre
is mapped one input data bit to one output sequence of length (spreadfactor). If there is
than one user specified, the next data bit is spread by the Walsh sequence corresponding
to the second user listed in the 'users' parameter, and so on until one data bit for
each user listed has been spread by the proper sequence.  These individual spread signals
then linearly summed and the composite signal is output.
       Note that since the incoming data stream is complex, the same
spreading sequence is applied to both the in-phase and quadrature channels.  This spreadin
consists of a real-valued scalar multiply of the complex data stream by the spreading
sequence.

        }
        version {@(#)SDFWalshInt2.pl    2.12 11/25/92}
        author { Craig Teuscher 5/93 }
        copyright {
Copyright (c) 1990, 1991, 1992 The Regents of the University of California.
All rights reserved.
See the file ~ptolemy/copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
        }
        location { user.pal }
        explanation {
        }

        hinclude {
            <math.h>
        }
        input {
                name { data }
                type { complex }
        }
        output {
                name { spreadout }
                type { complex }
        }
        defstate {
                name { users }
                type { IntArray }
                default {"0"}
                desc { The user numbers}
        }
        defstate {
                name { spreadfactor}
                type { int }
                default {"64"}
                desc { The spreading factor. }
        }
        protected {
               int walsh[128][128];
        }
        start {
               int i,j,r,bits,exponent;
               int userbits[8], timebits[8];
            bits =  (int) (log10(double(spreadfactor))/log10(2.0));

            for (i=0;i<users.size();i++) {
               converttobits(users[i],bits,userbits);
               userbits[bits]=0;
               for (j=0;j< int(spreadfactor);j++) {
                   exponent=0;
                   converttobits(j,bits,timebits);
                   timebits[bits]=0;
                   for (r=0;r<bits;r++) {
                       exponent += userbits[bits-1-r] * (timebits[r] + timebits[r+1]);
                   }
                   walsh[i][j] = (exponent%2==0) ? 1: -1;
               }
            }
            data.setSDFParams(users.size(),users.size()-1);
            spreadout.setSDFParams(int(spreadfactor),int(spreadfactor)-1);
        }
        go {
            int i,j;
            Complex sum;

            for (j=0;j<int(spreadfactor);j++) {
               sum=Complex(0,0);
               for (i=0;i<users.size();i++) {
                    sum += Complex(Complex(data%(users.size()-1-i)) * (float) walsh[i][j])
               }
               spreadout%(int(spreadfactor)-1-j) << sum;
            }
        }
        method {
              name { converttobits }
              access { protected }
              arglist {"(int number, int numberofbits, int *bits)" }
              type { void }
              code {
                   int i,divisor;

                   divisor=1;

                   for (i=0;i<numberofbits;i++) {
                       bits[i]= (number/divisor) %2;
                       divisor*=2;
                   }
              }
        }

}
```

```
defstar {
        name { PowerAGC }
        domain { CGC }
        desc {  This star models an AGC.  The signal arrives as a complex valued data
        stream.  The norm of the signal is averaged over the number of data points
        specified by the updategain parameter.  The gain is then updated (adjusted) so
        that the "power" of the signal has an average value equal to the specified
        powerlevel parameter.  The accumulator is reset and the process repeats.
        The output signal -- dataout -- consists of the input signal multiplied by the
        gain element.
        }
        version {@(#)CGCPowerAGC.pl     2.12 11/25/92}
        author { Craig Teuscher 2/93 }
        copyright {
Copyright (c) 1990, 1991, 1992 The Regents of the University of California.
All rights reserved.
See the file ~ptolemy/copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
        }
        location { user.pal }
        explanation {
        }
        input {
                name { data }
                type { complex }
        }
        output {
                name { dataout }
                type { complex }
        }
        output {
                name { gainout }
                type { float }
        }
        defstate {
                name { powerlevel }
                type { float }
                default { powerlevel }
                desc { The normalized powerlevel desired.}
        }
        defstate {
                name { updategain }
                type { int }
                default { updategain }
                desc { The number of samples to average before updating gain.}
        }

        codeblock (decls) {
              double $starSymbol(runningsum) = 0;
              int $starSymbol(counter) = 0;
              double $starSymbol(gain) = 1.0;
        }

        codeblock (mainblock) {
          complex temp;
          double temp1;

         /* Accumulate the norm of the signal (after it is multiplied by the present
            gain value). Increment the counter to keep track of the data points that
            have been processed. */

            temp.real=$starSymbol(gain)*$ref(data).real;
            temp.imag=$starSymbol(gain)*$ref(data).imag;
            $starSymbol(runningsum) += sqrt(temp.real*temp.real+temp.imag*temp.imag) /* no
            $starSymbol(counter)++;

        /* The gain value is output for convenience, while the dataout stream consists
           of the input stream multiplied by the gain value. */

            $ref(gainout)= $starSymbol(gain);
            $ref(dataout).real=temp.real;
            $ref(dataout).imag=temp.imag;

        /* If enough data points have been processed, update the gain so that the average
           powerlevel corresponds to the desired value.  Then reset the counter and accumu

            if ($starSymbol(counter)==$val(updategain)) {
               temp1=($val(powerlevel)*$starSymbol(counter))/$starSymbol(runningsum);
               $starSymbol(gain) *= sqrt(temp1);
               $starSymbol(counter)=0;
               $starSymbol(runningsum)=0;
            }
    }

    go {
       addInclude("<math.h>");
       addDeclaration(decls);
       addCode(mainblock);
    }
}
```

```
defstar {
    name { Walsh3 }
    domain { CGC }
    desc {
This star functions as a Walsh interference generator and data
spreader. The spreadfactor determines the length of the Walsh sequence
and should be of length (2^n) where n is an integer.  Each user is assigned
a unique Walsh sequence, and data bits are mapped into the appropriate
Walsh sequence by multiplying the sequence by +/- 1 depending on whether
the information bit is +/- 1. The 'users' parameter allows one to specify
all of the users in the system in one star. Simply list the user numbers
(ranging from 0 to (spreadfactor-1)) where 0 represents the pilot tone (DC)
channel.  For a single user, the incoming data stream is mapped one input data
bit to one output sequence of length (spreadfactor). If there is more
than one user specified, the next data bit is spread by the Walsh sequence
corresponding to the second user listed in the 'users' parameter, and so
on until one data bit for each user listed has been spread by the proper
sequence.  These individual spread signals are then linearly summed and the
composite signal is output.

Note that since the incoming data stream is complex, the same spreading
sequence is applied to both the in-phase and quadrature channels.
This spreading consists of a real-valued scalar multiply of the complex
data stream by the spreading sequence.

Beware: when using Complex-valued entities in CGC, you must explicitly
use conversion stars on either side of this block if you want only a real-
valued modulator.

The modonly state says that NO spreading should be done; spreadfactor in
this case is just the length of the Walsh sequence.  The output is simply
input*walsh value, one-to-one modulated.

In the case that modonly is one, then the stall input becomes active,
and allows clock synchronization between the PN sequence and the
Walsh mod sequence.
    }

    explanation {
This star functions as a Walsh interference generator and data
spreader. The spreadfactor determines the length of the Walsh sequence
and should be of length (2^n) where n is an integer.  Each user is assigned
a unique Walsh sequence, and data bits are mapped into the appropriate
Walsh sequence by multiplying the sequence by +/- 1 depending on whether
the information bit is +/- 1. The 'users' parameter allows one to specify
all of the users in the system in one star. Simply list the user numbers
(ranging from 0 to (spreadfactor-1)) where 0 represents the pilot tone (DC)
channel.  For a single user, the incoming data stream is mapped one input data
bit to one output sequence of length (spreadfactor). If there is more
than one user specified, the next data bit is spread by the Walsh sequence
corresponding to the second user listed in the 'users' parameter, and so
on until one data bit for each user listed has been spread by the proper
sequence.  These individual spread signals are then linearly summed and the
composite signal is output.

Note that since the incoming data stream is complex, the same spreading
sequence is applied to both the in-phase and quadrature channels.
This spreading consists of a real-valued scalar multiply of the complex
data stream by the spreading sequence.

Beware: when using Complex-valued entities in CGC, you must explicitly
use conversion stars on either side of this block if you want only a real-
valued modulator.
```

```
The modonly state says that NO spreading should be done; spreadfactor in
this case is just the length of the Walsh sequence.  The output is simply
input*walsh value, one-to-one modulated.  If modonly==1, no spreading;
modonly==0, will perform spread.

In the case that modonly is one, then the stall input becomes active,
and allows clock synchronization between the PN sequence and the
Walsh mod sequence.
    }

    author { Culprits: Craig Teuscher, CGC port and major modifications by Sam }

    input {
        name { data }
        type { complex }
    }

    input {
        name { stall }
        type { int }
    }

    output {
        name { spreadout }
        type { complex }
    }

    defstate {
        name { users }
        type { intarray }
        default {"20"}
        desc { The Walsh number(s) to use in the spreading.  This is an array parameter.
    }

    defstate {
        name { spreadfactor}
        type { int }
        default {"64"}
        desc { Spreading factor }
    }

    defstate {
        name { modonly }
        type { int }
        default {"0"}
        desc { Modulate only? }
    }

    setup {
        data.setSDFParams(users.size(),users.size()-1);
        if (int(modonly)==0) {
           spreadout.setSDFParams(int(spreadfactor),int(spreadfactor)-1);
        }
    }

    initCode {
        if (int(modonly)==1) {
            addDeclaration("int $starSymbol(index) = 0;");
        }
        addInclude("<math.h>");
        if (addGlobal(walsharr,"$sharedSymbol(Walsh2,Walsh2)")) {
            addCode(sharedinit);
        }
    }
    go {
        if (int(modonly)==0) {
            addCode(maincode);
        } else if (int(modonly)==1) {
            addCode(maincodeMod);
        }
    }

    codeblock(maincode) {
        int i,j;
        complex sum;

        for (j=0;j<$val(spreadfactor);j++) {
           int outcount = $val(spreadfactor)-j-1;

           sum.real = 0.0;
           sum.imag = 0.0;
           for (i=0;i<$size(users);i++) {
                int dataoffset = $size(users) - 1 -i;
                sum.real += $ref(data,dataoffset).real *
                            (float) $sharedSymbol(Walsh2,walsh)[j][$ref(users)[i]];
                sum.imag += $ref(data,dataoffset).imag *
                            (float) $sharedSymbol(Walsh2,walsh)[j][$ref(users)[i]];
           }
           $ref(spreadout,outcount).real = sum.real;
           $ref(spreadout,outcount).imag = sum.imag;
        }
    }

    codeblock(maincodeMod) {
        int i;
        complex sum;

        sum.real = 0.0;
        sum.imag = 0.0;
        for (i=0;i<$size(users);i++) {
            int dataoffset = $size(users) - 1 -i;
            sum.real += $ref(data,dataoffset).real *
                (float) $sharedSymbol(Walsh2,walsh)[$starSymbol(index)][$ref(users)[i]];
            sum.imag += $ref(data,dataoffset).imag *
                (float) $sharedSymbol(Walsh2,walsh)[$starSymbol(index)][$ref(users)[i]];
        }

        /* Deal with the stall case on the Walsh mod */

        if ($ref(stall) != 1) {
            $starSymbol(index) = (++$starSymbol(index)) % $val(spreadfactor);
        }

        $ref(spreadout).real = sum.real;
        $ref(spreadout).imag = sum.imag;
    }


    codeblock(walsharr) {
        char $sharedSymbol(Walsh2,walsh)[$val(spreadfactor)][$val(spreadfactor)];

        void $sharedSymbol(Walsh2,ConBits)(int number, int numberofbits, int *bits)
        {
            int i;

            for (i=0;i<numberofbits;i++) {
                bits[i]= (number >> i) & 0x01;
            }
            bits[numberofbits]=0;
        }
    }

    codeblock(sharedinit) {
       {
           int i,j,r,bits,exponent;
           int *userbits, *timebits;


           bits = (int) ceil(log2((double) $val(spreadfactor)));
           userbits = (int *) calloc(bits+1, sizeof(int));
           timebits = (int *) calloc(bits+1, sizeof(int));

           for (i=0; i < $val(spreadfactor); i++) {
                $sharedSymbol(Walsh2,ConBits)(i,bits,userbits);

                for (j=0;j< $val(spreadfactor);j++) {
                    exponent=0;
                    $sharedSymbol(Walsh2,ConBits)(j,bits,timebits);

                    for (r=0;r<bits;r++) {
                        exponent += userbits[bits-1-r] *
                            (timebits[r] + timebits[r+1]);
                    }
                    /* For memory efficiency, we mark this as [j][i] */
                    $sharedSymbol(Walsh2,walsh)[j][i]  = (exponent%2==0) ? 1 : -1;
               }
           }
       }
    }
}
```

```
defstar {
    name { WalshCorr7 }
    domain { CGC }
    desc {
This star functions as a Walsh interference generator and data
spreader. The spreadfactor determines the length of the Walsh sequence
and should be of length (2^n) where n is an integer.  Each user is assigned
a unique Walsh sequence, and data bits are mapped into the appropriate
Walsh sequence by multiplying the sequence by +/- 1 depending on whether
the information bit is +/- 1. The 'users' parameter allows one to specify
all of the users in the system in one star. Simply list the user numbers
(ranging from 0 to (spreadfactor-1)) where 0 represents the pilot tone (DC)
channel.  For a single user, the incoming data stream is mapped one input data
bit to one output sequence of length (spreadfactor). If there is more
than one user specified, the next data bit is spread by the Walsh sequence
corresponding to the second user listed in the 'users' parameter, and so
on until one data bit for each user listed has been spread by the proper
sequence.  These individual spread signals are then linearly summed and the
composite signal is output.

Note that since the incoming data stream is complex, the same spreading
sequence is applied to both the in-phase and quadrature channels.
This spreading consists of a real-valued scalar multiply of the complex
data stream by the spreading sequence.

Beware: when using Complex-valued entities in CGC, you must explicitly
use conversion stars on either side of this block if you want only a real-
valued modulator.

The modonly state says that NO spreading should be done; spreadfactor in
this case is just the length of the Walsh sequence.  The output is simply
input*walsh value, one-to-one modulated.

In the case that modonly is one, then the stall input becomes active,
and allows clock synchronization between the PN sequence and the
Walsh mod sequence.
    }

    explanation {
This star functions as a Walsh interference generator and data
spreader. The spreadfactor determines the length of the Walsh sequence
and should be of length (2^n) where n is an integer.  Each user is assigned
a unique Walsh sequence, and data bits are mapped into the appropriate
Walsh sequence by multiplying the sequence by +/- 1 depending on whether
the information bit is +/- 1. The 'users' parameter allows one to specify
all of the users in the system in one star. Simply list the user numbers
(ranging from 0 to (spreadfactor-1)) where 0 represents the pilot tone (DC)
channel.  For a single user, the incoming data stream is mapped one input data
bit to one output sequence of length (spreadfactor). If there is more
than one user specified, the next data bit is spread by the Walsh sequence
corresponding to the second user listed in the 'users' parameter, and so
on until one data bit for each user listed has been spread by the proper
sequence.  These individual spread signals are then linearly summed and the
composite signal is output.

Note that since the incoming data stream is complex, the same spreading
sequence is applied to both the in-phase and quadrature channels.
This spreading consists of a real-valued scalar multiply of the complex
data stream by the spreading sequence.

Beware: when using Complex-valued entities in CGC, you must explicitly
use conversion stars on either side of this block if you want only a real-
valued modulator.
```

```
The modonly state says that NO spreading should be done; spreadfactor in
this case is just the length of the Walsh sequence.  The output is simply
input*walsh value, one-to-one modulated.  If modonly==1, no spreading;
modonly==0, will perform spread.

In the case that modonly is one, then the stall input becomes active,
and allows clock synchronization between the PN sequence and the
Walsh mod sequence.
    }

    author { Culprits: Craig Teuscher, CGC port and major modifications by Sam }

    input {
        name { data }
        type { complex }
    }

    input {
        name { stall }
        type { int }
    }

    output {
        name { spreadout }
        type { complex }
    }

    defstate {
        name { user }
        type { int }
        default {"20"}
        desc { The Walsh number to use in decorrelating. }
    }

    defstate {
        name { spreadfactor}
        type { int }
        default {"64"}
        desc { Spreading factor }
    }

    setup {
        data.setSDFParams(int(spreadfactor),int(spreadfactor)-1);
        stall.setSDFParams(int(spreadfactor),int(spreadfactor)-1);
    }

    initCode {
        addDeclaration("int $starSymbol(index) = 0;");
        addInclude("<math.h>");
        if (addGlobal(walsharr,"$sharedSymbol(Walsh2,Walsh2)")) {
            addCode(sharedinit);
        }
    }
    go {
            addCode(maincode);
    }

    codeblock(maincode) {
        int i,j;
        static complex sum;
        int dataoffset=0;
        int dumped=0;

        for (j=0;j<$val(spreadfactor);j++) {
          dataoffset=$val(spreadfactor)-1-j;
          if ($starSymbol(index)==0) {  /*Reset summation to zero*/
              sum.real=0;
              sum.imag=0;
              }

          sum.real += $ref(data,dataoffset).real *
                            (float) $sharedSymbol(Walsh2,walsh)[$starSymbol(index)][$val(u
          sum.imag += $ref(data,dataoffset).imag *
                            (float) $sharedSymbol(Walsh2,walsh)[$starSymbol(index)][$val(u

          if ($starSymbol(index)==$val(spreadfactor)-1) {        /* dump data if end of se
              $ref(spreadout).real = sum.real;
              $ref(spreadout).imag = sum.imag;
              dumped=1;
              }

      /* Handle stall case */
          if ($ref(stall,dataoffset) != 1) $starSymbol(index) = ++$starSymbol(index) % $v
          }
      if (dumped==0) {  /* if haven't dumped, output a zero */
         $ref(spreadout).real = 0.0;
         $ref(spreadout).imag = 0.0;
         }
    }

    codeblock(walsharr) {
        char $sharedSymbol(Walsh2,walsh)[$val(spreadfactor)][$val(spreadfactor)];

        void $sharedSymbol(Walsh2,ConBits)(int number, int numberofbits, int *bits)
        {
            int i;

            for (i=0;i<numberofbits;i++) {
                bits[i]= (number >> i) & 0x01;
            }
            bits[numberofbits]=0;
        }
    }

    codeblock(sharedinit) {
      {
          int i,j,r,bits,exponent;
          int *userbits, *timebits;


          bits = (int) ceil(log2((double) $val(spreadfactor)));
          userbits = (int *) calloc(bits+1, sizeof(int));
          timebits = (int *) calloc(bits+1, sizeof(int));

          for (i=0; i < $val(spreadfactor); i++) {
               $sharedSymbol(Walsh2,ConBits)(i,bits,userbits);

               for (j=0;j< $val(spreadfactor);j++) {
                   exponent=0;
                   $sharedSymbol(Walsh2,ConBits)(j,bits,timebits);

                   for (r=0;r<bits;r++) {
                       exponent += userbits[bits-1-r] *
                            (timebits[r] + timebits[r+1]);
                   }
                   /* For memory efficiency, we mark this as [j][i] */
                   $sharedSymbol(Walsh2,walsh)[j][i]  = (exponent%2==0) ? 1 : -1;
              }
            }
        }
    }
}
```