Asynchronous Receivers in

Narrowband Packet Radio Applications

by

Amit G. Bagchi

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 23, 1997

Author _____

Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by _____

Amos Lapidoth
Thesis Supervisor

Accepted by _____

F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Asynchronous Receivers in Narrowband Packet Radio Applications:
Personal Access Communications System (PACS) and the U.S. Asynchronous Band

by

Amit G. Bagchi

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 1997

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

ABSTRACT

A burst coherent receiver capable of operating efficiently in packet radio applications is studied. Rules for use of the U.S. Asynchronous spectrum are discussed, with attention to their implications for receiver design. An existing design by Bell Communications Research is reviewed in the context of packet radio applications and problems with this design that lead to low spectral efficiency and low throughput are noted. Two major problems are found to be excessive synchronization overhead and excessive processing latency. A technique for reducing latency by variable rate processing is described in detail, with analysis of achievable reductions provided. Methods for reducing synchronization overhead are proposed, and a simulation environment for evaluating their performance is presented. Finally, limited simulation results are provided for initial performance estimates.

Thesis Supervisor: Amos Lapidoth
Title: Assistant Professor, Department of Electrical Engineering and Computer Science

Table of Contents

Table of Figures

1. Acknowledgement

This work would not have been possible without the valuable assistance of Dr. Robert Ziegler, Director of the Wireless Enterprise Systems Group at Bell Communications Research, Inc. (Bellcore). His initial development of MATLAB simulations for the PACS demodulator, as well has his comments and suggestions are the foundation for this work. Bellcore Research Scientist Gregory Pollini was instrumental in formulating the latency of the PACS receiver, and my collaboration with him is included here. All of the members of the Wireless Enterprise Systems Group and the Wireless Techniques and Technologies Group, have been very supportive in the course of this work. Finally, Massachusetts Institute of Technology Professor Amos Lapidoth, provided guidance as my thesis advisor.

2. Introduction

2.1 Motivation

In recent years the U.S. has experienced a fast growing demand for tetherless data capability, wireless local area networks, and mobile networking capability. This, coupled with the recent allocation of U.S. spectrum for such use, has led many to develop these systems. However, relatively high bit error rates are tolerated in developing the required physical layer hardware. The motivation for exploring burst coherent receivers for these applications, is the lower bit error rates that they provide. Bell Communications Research has designed a burst coherent receiver suitable for circuit switched telephony with the Personal Access Communications System standard. However, the higher processing latency associated with this receiver renders it inefficient in packet switched applications. It is the author's intention to design a coherent receiver suitable for packet radio applications with relatively low bit error rates.

2.2 U.S. Spectrum for Asynchronous Use

In order to better understand the design requirements for the desired receiver, the target spectrum must be discussed. The United States Federal Communications Commission (FCC) has allocated the electromagnetic spectrum between 1850 and 1990 MHz for the delivery of "Personal Communications Services." It is referred to as the Emerging Technologies Spectrum for Broadband PCS, and it is subdivided into licensed and unlicensed bands. In the licensed band, providers have purchased the rights to spectrum at auction, whereas in the unlicensed band, any device may operate that conforms to FCC specified usage rules. The unlicensed band is further subdivided into

Isochronous and Asynchronous bands. The FCC Isochronous Band rules enable telephony applications such as wireless PBXs, while the Asynchronous rules enable packet radio applications such as wireless LANs. It is the availability of the Unlicensed Asynchronous Spectrum between 1910 and 1930 MHz that motivates the development of asynchronous receivers.

2.3 Personal Access Communications System (PACS)

PACS is a North American standard for low power digital wireless communications. It uses frequency division multiplexing to separate transmissions from different radio access ports (RPs), and time division multiplexing to separate the transmissions of multiple subscriber units to a single RP. It enables low complexity implementation of radio transceivers by emphasizing low transmit power levels, relatively small RP coverage areas (as compared to cellular base stations), and appropriately low link transmission rates. PACS is inherently suited for isochronous telephony applications. However, in its direct application to asynchronous data, too much transmission overhead would be required for the receiver to synchronize with the transmitter at the physical layer. In addition, Bellcore's PACS receiver has too much latency to permit rapid responses to short transmissions.

2.4 Packet Data Applications

In packet data applications, such as wireless LANs, peak throughput is reduced by this overhead, especially as the size of individual packet transmissions approaches the burst size over which coherent demodulation is performed. Furthermore, throughput is reduced when an asynchronous network of devices is unable to keep possession of the

unlicensed spectrum for sustained periods of time. FCC rules require cooperating devices to surrender the spectrum if they fail to transmit for a period exceeding 25 us. The gap between asynchronous PACS transmissions is likely to exceed the specified time due to a combination of the FCC listen before talk etiquette and the long PACS receiver latency. If the minimum transmission size approaches the length of receiver latency, then this problem is avoided. However, the transmission of such a long packet, over a wireless channel, without physical layer error correction, would most likely itself reduce throughput with retransmission traffic.

Thus, two parameters of the current PACS receiver limit peak throughput in the asynchronous spectrum. First, excessive transmission overhead is required for receiver synchronization. Second, excessive receiver latency implies that either a given network has only limited access to the spectrum, or that to keep access to the spectrum the minimum packet length is restricted. The goal of this work is to determine methods that reduce receiver overhead and latency, thereby increasing peak throughput.

3. U.S. Asynchronous Spectrum Rules

The rules for the use of the 1910 - 1930 MHz spectrum are specified in FCC Part 15 Subpart D. The 1910 - 1920 MHz band is specifically set aside for use by asynchronous devices. ANSI rules specify measurement procedures for device conformance. Together, these rules constraint the receiver design significantly. Discrepancies between the two sets of rules create ambiguity, however an interpretation of their intent follows.[1]

FCC Rule	*Description*
15.321a	**Channel Allocation.** The emission bandwidth of any transmitter must be at least 500 kHz.
15.321b	**Channel Selection.** Users with a channel bandwidth less than 2.5 MHz begin searching for an available channel in the 3 MHz at either end of the band (1910-1913, 1917-1920 MHz). Users with a channel bandwidth greater than 2.5 MHz start searching in the center half of the band (1912.5-1917.5 MHz). Devices with a bandwidth less than 1 MHz may not occupy the center half of the band if spectrum is available elsewhere.
15.321c.1	**Monitoring Time.** Before transmitting the user must monitor the channel for at least 50 us. 15.321c.3 specifies that a user or cooperating group of users does not need to monitor the channel given that the inter-burst gap requirement is met.
15.321f	**Inter-burst Gap.** The inter-burst gap between transmissions of cooperating devices cannot exceed 25 us.
15.321c.4	**Random Waiting.** After each transmission, a user must "wait a deference time randomly chosen from a uniform random distribution ranging from 50 to 750 microseconds..." If an access attempt fails "the range of the deference time chosen shall double until an upper limit of 12 milliseconds is reached. The deference time remains at the upper limit of 12 milliseconds until an access attempt is successful. The deference time is re-initialized after each successful access attempt."
15.321f	**Maximum Transmit Period.** The transmission from a device or group of cooperating devices must be less than 10 ms.

Table 3-1 Interpretation of FCC Rules for Asynchronous Spectrum

4. PACS Design and the Asynchronous Spectrum

In modifying the existing PACS receiver, it is necessary to revisit the engineering assumptions that are adopted in its design. Many of these assumptions remain valid, as do the resulting design decisions. Basic elements such as the channel model, the modulation method, error check codes, the symbol rate and the demodulation method characterize the physical layer design. These elements must be reviewed.

4.1 Channel Model

As typical in PACS system engineering, a flat fading channel model is assumed. Flat fading implies that the channel frequency response is flat over the range of interest, and fading involves the movement of this flat level over time. Fading is assumed to occur at a rate such that channel characteristics do not significantly change over the time interval of a received burst. The channel frequency response is flat (limited intersymbol interference) because the RP coverage area is small, and transmit power levels are low.[2]

4.2 Modulation

The PACS modulation method is $\pi/4$ shifted DQPSK (Differential Quaternary Phase Shift Keyed). Phase modulation is chosen since amplitude information is more vulnerable to fading environments, and allows hard limiting in the receiver, reducing complexity. Quaternary phase is chosen for its capacity increase over BPSK, 2 bits per symbol as opposed to 1 bit per symbol, and increased robustness over higher order PSK. Finally, $\pi/4$ shifted DQPSK is chosen to guarantee phase changes every symbol, and prevent phase transitions through the origin of the complex phase plane.[3] Phase transitions through the origin can occur in other systems when a phase change of $+/- \pi$ is made. Figure 4-1 depicts the differential and absolute phase constellations of the modulation scheme. The differential constellation on the left shows the bit encoding of each symbol, and the absolute constellation on the right shows the resulting possible absolute phases, and possible transitions associated with each symbol. None of these design decisions are changed by asynchronous operation.

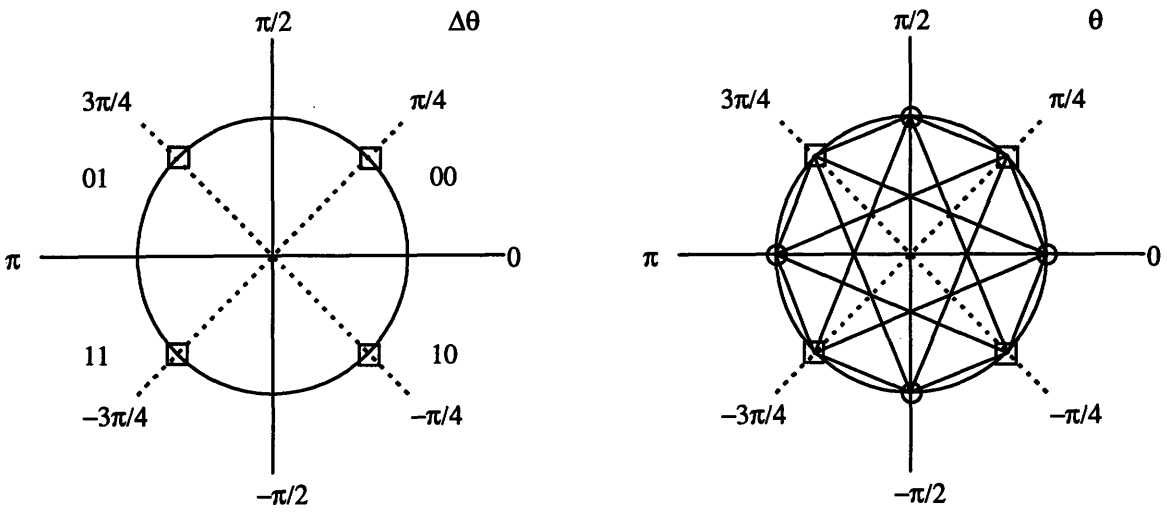


Figure 4-1 Differential (left) and absolute (right) phase constellations for π/4 shift DQPSK

4.3 Error Check Coding

The error coding of the asynchronous transmitter is assumed to conform to the PACS standard. The PACS standard employs a cyclic redundancy check, essentially a parity check code. Work by Bellcore has determined that given the channel model assumed, more sophisticated means of error correction offer little performance gain.[4] The rules for choosing the optimal CRC code for PACS, using a 105 bit code word, 90 bit data and 15 bit CRC, are not affected by simply moving from isochronous to asynchronous operation. However, if higher layer protocol work by others indicates that a different packet size is desirable for asynchronous operation, a new CRC code must be chosen.

4.4 Burst Size / Symbol Rate

As described earlier, PACS employs time division multiplexing to distinguish between multiple subscriber unit transmissions to a single RP. Thus, there is already a

concept of burst size in the PACS transceiver, since subscriber units transmit and receive according to a fixed, repeating time schedule called a frame. A PACS burst is 54 symbols in duration, or 106 bits, with successive bursts separated by 6 symbols of guard time. The 106 bits includes the 105 bit codeword and a pad bit. In order to preserve as many elements of the existing receiver as possible, the 60 symbol PACS burst will be considered an atomic unit.

One parameter that must change to satisfy the FCC rules is the symbol rate. The standard PACS symbol rate is 192 kHz. Using a square root raised cosine transmit pulse shape with a roll off factor of 0.5, the transmission occupies a band at carrier +/- 1.5*96 kHz, a bandwidth of 288 kHz. The FCC rules require that any intentional transmitter use at least 500 kHz of bandwidth. The easiest way to map PACS into compliance is to double the symbol rate, occupying a bandwidth of 576 kHz. While it may be desirable to increase the symbol rate further, occupying more spectrum, it is more difficult to implement such a design.

4.5 Coherent Demodulation

The existing PACS receiver uses coherent methods with good error performance in multipath fading environments. The PACS radio port has a free running oscillator that determines the carrier frequency. The PACS terminal receiver adjusts its oscillator to maintain phase lock with the radio port. The terminal receiver estimates the absolute phase of the transmission and performs carrier and phase recovery on the phase stream.

The current receiver uses digital block processing techniques to jointly estimate the symbol timing and carrier offset on a burst-by-burst basis. The symbol timing estimate is

made by accumulating a metric representing eye opening over a burst for all candidate sampling phases in the over-sampled receive stream. The carrier offset estimate is made by calculating the rotation of the differential phase constellation. The estimation window is 38 symbols long, a majority of the burst. Carrier frequency offset and phase offset estimates are further refined with a second order, critically damped loop. The differential phase information is extracted only after frequency and phase compensation of the absolute phase.[5]

The advantage of this coherent method is reduced sensitivity to local symbol impairments by averaging and accumulation over burst long periods. When fading occurs fast enough to change channel characteristics within a given burst, non-coherent methods become more attractive. However, for the channel model assumed, coherent receivers retain their advantage.

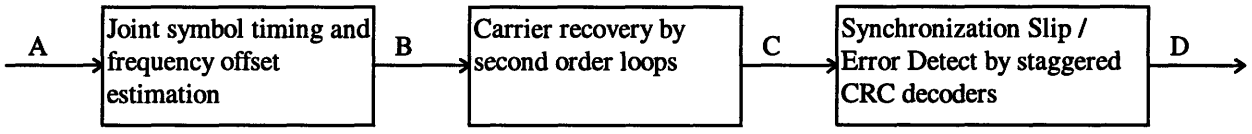
5. Problems of the Existing PACS Receiver

5.1 Transmission Overhead and Minimum Packet Size

Since the existing PACS receiver employs block processing methods, several throw-away bursts are required before the receiver can synchronize with the transmission. The receive window must significantly overlap a valid transmission for the estimated parameters of symbol timing and frequency offset to be valid. During the first burst these parameters are estimated in the symbol timing process, during the second, carrier recovery converges on finer grain phase and frequency offsets and reconstructs symbols, and during the third, a CRC error check is performed. Figure 5-1 shows these 3 stages and the input and output streams at each stage. In the downlink direction (RP to terminal), correlation

to a unique bit sequence, before the CRC check, provides a reference point for synchronization. The CRC check cannot succeed unless the receiver has determined the start of the protected code-word in the transmission, so synchronization is necessary before any data can be successfully demodulated *and* decoded. Figure 5-2 assumes that the unique bit sequence is at the start of each transmission. Note that for one burst of data to be sent, 4 bursts of overhead must precede it. This is unacceptable in packet radio applications.

The reason for the receiver's long latency is its emphasis on low power consumption and slow clocks. The speed of the parameter estimation operation is limited by the speed of the incoming transmission, but subsequent carrier recovery and decoding are performed at the symbol and bit rates. By performing these operations at a faster rate, latency can be reduced.



A: Over sampled IF stream (20 samples per symbol)
B: Absolute Phase stream at symbol rate
C: Symbol stream at symbol rate (2 bits per symbol)
D: Error checked bit stream at bit rate (2x symbol rate)

Figure 5-1 Receiver Pipeline Stages

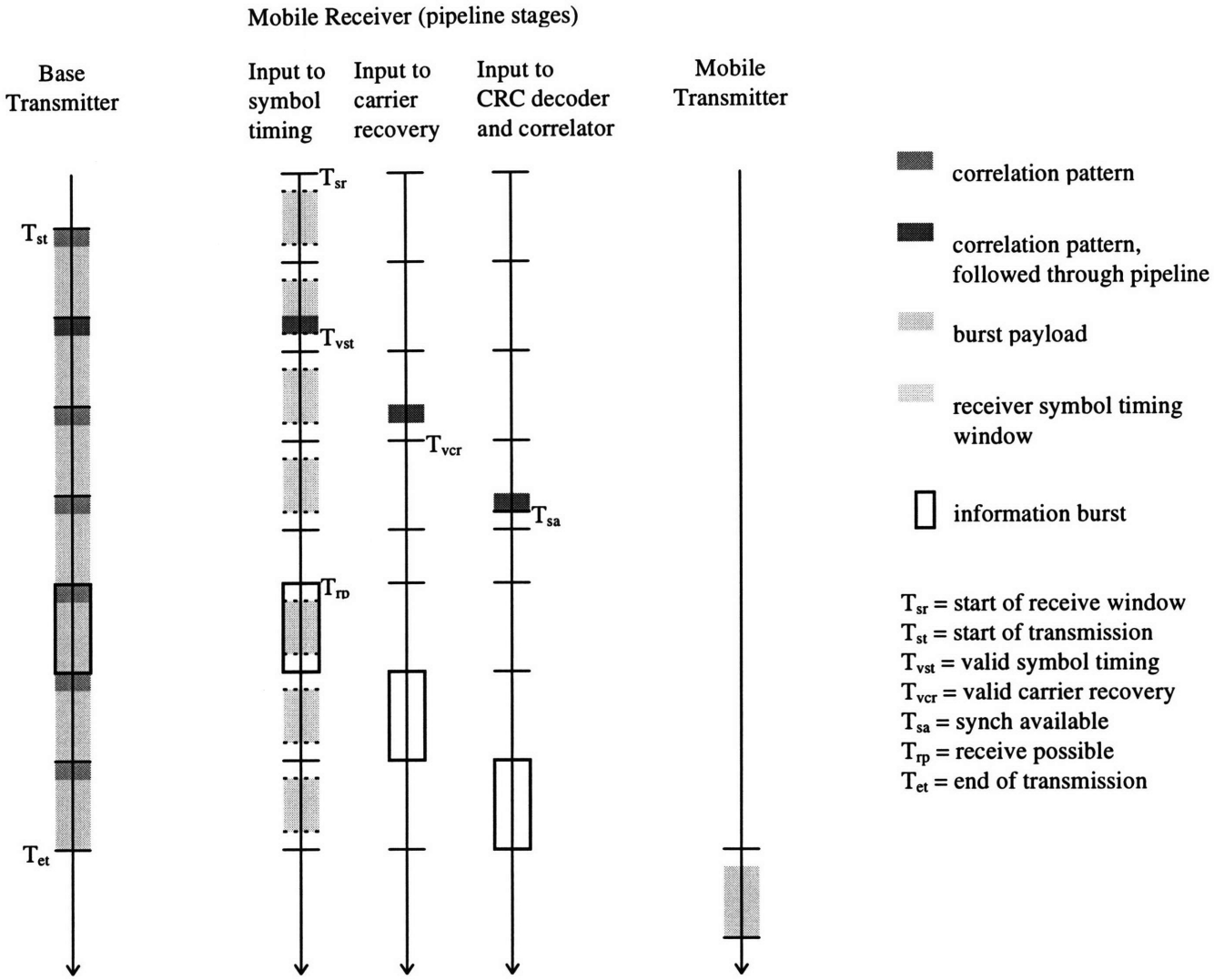


Figure 5-2 Overhead requirements of current design.

5.2 Demodulation / Decode Latency and Terminal Response Time

The receiver latency poses another problem when viewed in the context of the Asynchronous spectrum etiquette. The inter-burst gap requirement specifies that the gap between successive transmissions of cooperating devices is less than 25 us. Given that a transmission occurs as depicted in Figure 5-2, the spectrum must remain occupied until the terminal is able to respond. Due to approximately three bursts of latency, data is not

available to the terminal until two bursts after the transmission has ceased. As a result the base must transmit 2 wasted bursts after the information burst. Note that 7 bursts had to be transmitted to send one burst of information.

Note that there are two related problems, excessive processing latency and excessive overhead. However, even if latency is drastically reduced, the limiting case on overhead is still one burst. One burst of overhead is required to ensure that a valid transmission overlaps the symbol timing estimation window. So some novel approach is required at the symbol timing stage.

6. New Receiver Techniques

6.1 Media Access Control Perspective

The challenging environment of an asynchronous data network requires a receiver with low latency and low synchronization overhead. For this reason, the network design decisions made in media access control (MAC), determine the requirements imposed on the receiver. The FCC rules for cooperating devices constrain MAC layer design, and in turn, receiver design. Two MAC layer methods for meeting the inter-burst gap rule impose very different requirements on receiver latency. Furthermore, different scenarios for RP / terminal cooperation lead to different overhead requirements.

Two approaches to meeting the 25 us inter-burst gap rule impose weak and strong constraints on latency. The first approach is to allow any two terminals to act as a cooperating group. Consequently, each terminal's transmissions must be separated by less

than 25 us. This requires a given terminal receiver to be able to demodulate and decode a transmission within 1 burst and 25 us, with enough time remaining for the terminal to recognize an identifying user address and possibly initiate a transmission, all before the 25 us has expired. A second approach is to employ a polling scheme where the base sends a series of packets, each of which requires a response from a different mobile terminal.[6] Given a system of a base and 3 terminals, let 3 packets be p_1, p_2, and p_3, and three mobile terminals be m_1, m_2, m_3, as depicted in Figure 6-1. The base first transmits some required overhead followed by packets p_1 through p_3. Assuming a receiver latency of approximately 3 bursts, packet p_1 will be decoded by mobile m_1 while the base is transmitting packet p_3. The mobile can then send response r_1. During response r_1, mobile m_2 decodes packet p_2 and is ready to respond with r_2 within the following 25 us. In this manner an arbitrary number of packets can be "piggybacked" to work around any hardware latency. However, it is desirable to minimize the number of piggybacked transmissions, since the failure of any given mobile terminal to respond breaks the chain and violates the 25 us constraint, allowing other groups to seize the spectrum. From this example one can conclude that the length of the polling chain is the smallest integer number of bursts greater than the receiver latency, or $\lceil latency_{(bursts)} \rceil$. So even if a polling scheme is employed, it is desirable to minimize the length of the polling chain, and consequently, the receiver latency. Note that this imposes a weaker constraint on latency than the non-polling approach, since any latency between n and n + 1 bursts in duration, where n is an integer, still leads to a polling chain of n + 1 terminals.

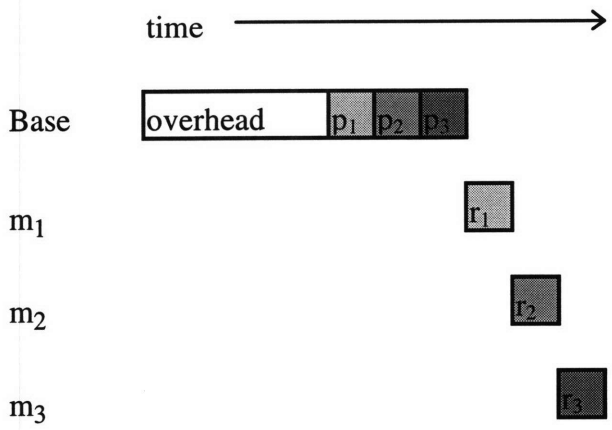


Figure 6-1 Polling Access piggybacking transmissions

Different scenarios for device cooperation also impact the overall overhead to payload ratio. While the spectrum usage is, in general, asynchronous, a cooperating group can use it in a synchronous, slotted manner during its possession. For example, one could require the base to be the first transmitter in the cooperating group. The base would gain spectrum access asynchronously, but all cooperating mobile terminals could then synchronize to the base's transmission, and communicate synchronously in burst-size slots for up to 10 ms. While there may be carrier and clock offsets between different terminals, they are unlikely to drift out of synchronization within 10 ms (64 bursts), and there are digital phase locked loop methods from PACS telephony that would keep them synchronized. As a result, the transmission overhead required for synchronization only occurs at the start of the group's exchange. The longer the exchange lasts, that is the closer to 10 ms duration, the lower the overhead ratio. As a result, it may not be worthwhile to decrease synchronization overhead only slightly, at the cost of increasing complexity significantly.

6.2 Reducing Latency

6.2.1 Variable Rate Processing

Upon consideration of the existing pipelined receiver it has been determined that some modules can operate at much higher clock rates, and some of the pipelined modules can begin processing before previous stages are done. Together these two techniques can yield significant latency reduction.

The effect of these techniques can be depicted using a two dimensional graph. On the vertical axis is time, measured in channel symbols, increasing downward. On the horizontal axis is the symbol number in a given burst that is being processed, increasing to the right. Therefore, solid lines depict the progress of each stage in processing the burst. The closer the slope of a given line to being horizontal, the higher the processing speed of that stage.

First, consider the performance of the existing design. Table 6-1 describes significant times in the processing of the received stream. It also characterizes the type of inputs entering each stage, as operations transform the data stream.

Time	*Description*
T_{A1}	**Start of oversampled IF data** entering receiver, where high pass filtering, digital mixing and bandpass to phase conversion occurs.
T_{A2}	**Start of oversampled phase stream** entering symbol timing estimation, where eye opening metric for each sampling position is accumulated over window.
T_{A3}	**Symbol timing estimates** of best sampling position and frequency offset corresponding to sampling position with greatest eye opening metric are available.
T_{A4}	**Phase stream downsampled** to symbol rate at estimated symbol timing position.

T_{A5}	**End of oversampled phase stream** entering symbol timing estimation.
T_{B1}	**Start of chosen phases** at symbol rate entering carrier recovery, forward compensation loop initialized with frequency offset estimate.
T_{B2}	Finer grain frequency and phase offset **estimates convergent**, backward compensation loop initialized, forward loop continues
T_{B3}	**Carrier recovery complete**, 2 bit symbols serialized into bit stream at bit rate
T_{C1}	**Bit stream delayed** by 6 bits (3 symbols) enters CRC decoder, delay is nominal, allows compensation for slips of the actual position of codeword -6 bits, -4 bits, -2 bits, 0 bits, +2 bits, +4 bits, +6 bits
T_{C2}	**Demodulated, decoded bit stream** or indication of error is available

Table 6-1 Reference time labels in demodulation and decoding.

Figure 6-2 illustrates the latencies of each stage, using the reference time labels. Note that the symbol timing estimate used to downsample the oversampled phase stream is available at time T_{A3}, well before it is used at time T_{A4}. As a result, carrier recovery could start earlier. Also, note that all the data to be processed by the carrier recovery and CRC decoding stages (T_{B1} to T_{C2}) is available at time T_{A5}. Therefore, while the symbol timing stage is constrained by the channel symbol rate, the carrier recovery and decoding stages can be run at an arbitrarily high multiple of the symbol rate, *r*. This is represented graphically by flattening the slope of the carrier recovery and decoding lines. The case where *r* is 2 is depicted on the right of Figure 6-2. The benefit of increasing *r* is constrained by the fact that the carrier recovery and decoding stages cannot process data before its available. This violation could be represented by allowing the carrier recovery line to cross the symbol timing line. Thus, time T_{A5} imposes a limit on the minimum latency.

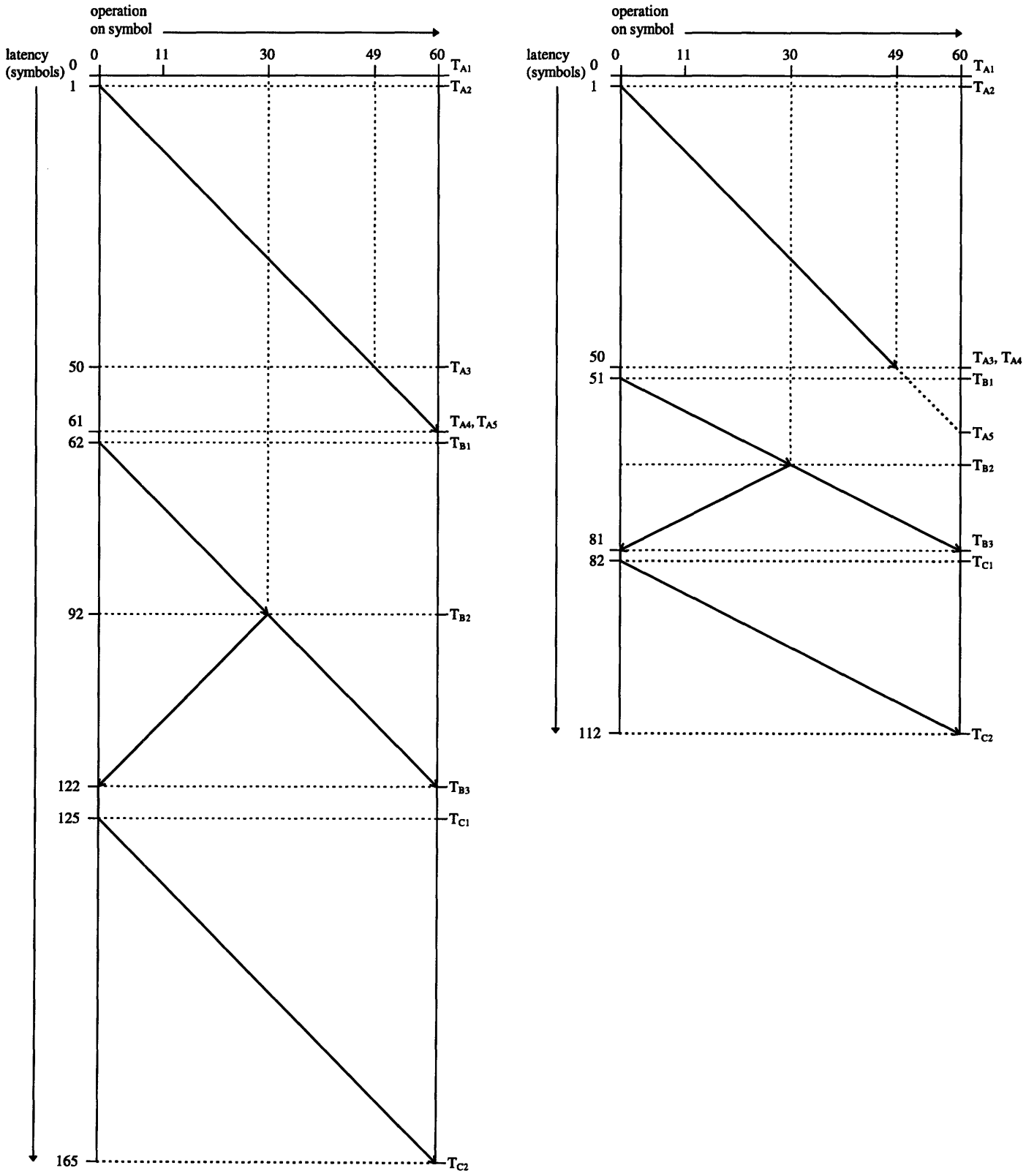


Figure 6-2 Existing design (left), and variable processing improvement (right).

The following derivation, developed with Research Scientist, Gregory Pollini, quantifies the minimum bound on the latency of the current receiver, expressed in terms of the channel symbol rate and *r*.

Let the reference time T_{A1} equal 0. Let the duration of one burst, or 60 symbols, equal T. The filtering and mixing operations that follow are implemented with a 5 tap FIR filter, and a set of look up tables for trigonometric functions, all at the oversampled rate. The latency of the filtering, mixing and conversion to phase operations is therefore less than one symbol. To obey symbol boundaries, it is most convenient to round the delay to 1 channel symbol, so that $T_{A2} = \frac{1}{60}T$. The symbol timing metric accumulation window is 38 symbols long, starting 11 symbols into the burst, so $T_{A3} = \frac{50}{60}T$. Since the burst is 60 symbols long, T_{A5} is 60 symbols after T_{A2}, so $T_{A5} = \frac{61}{60}T$.

The main distinction between the existing design and the improved design is that the carrier recovery process can start as early as a symbol delay after time T_{A3}, when the estimates used to initialize it are ready. This symbol delay is incurred in the phase stream downsampling process. Thus, in the improved design, $T_{A4} = T_{A3} = \frac{50}{60}T$, and $T_{B1} = \frac{51}{60}T$. Assuming the most general case, in which *r* is an arbitrarily high multiplier, the carrier recovery process could overtake symbol timing. The processing rate of carrier recovery would then have to slow down to the rate of symbol timing to avoid overtaking

it. So $T_{B3} = \max\left[T_{B1} + \frac{1}{r}T, \frac{62}{60}T\right]$. The limit imposed by T_{A5} is now apparent, as $\frac{62}{60}T$ is the one symbol downsampling delay after T_{A5}.

Decoding can begin 3 symbols after the backward compensation loop is finished. Since the backward loop is processing stored phases from the first half of the burst, it is not limited by T_{A5}, allowing decoding to begin at $T_{C1} = T_{B1} + \frac{1}{r}T + \frac{3}{60r}T$. There is a possibility that decoding could also overtake symbol timing, so T_{C2} is limited by T_{A5} such that $T_{C2} = \max\left[51 + \frac{123}{r}, 62 + \frac{3}{r}\right]$. Figure 6-3 summarizes these results and expresses time in terms of channel symbols.

$$T_{A2} = 1$$
$$T_{A3} = T_{A4} = 50$$
$$T_{B1} = 51$$
$$T_{A5} = 61$$
$$T_{B3} = \max\left[51 + \frac{60}{r}, 62\right]$$
$$T_{C1} = 51 + \frac{63}{r}$$
$$T_{C2} = \max\left[51 + \frac{123}{r}, 62 + \frac{3}{r}\right]$$

Figure 6-3 Latencies with variable rate processing

The final expression for the minimum latency of the current receiver using variable processing rates is $T_{C2} = \max\left[51+\frac{123}{r}, 62+\frac{3}{r}\right]$ symbols. The dependence of latency on the rate boost r is depicted graphically in Figure 6-4.

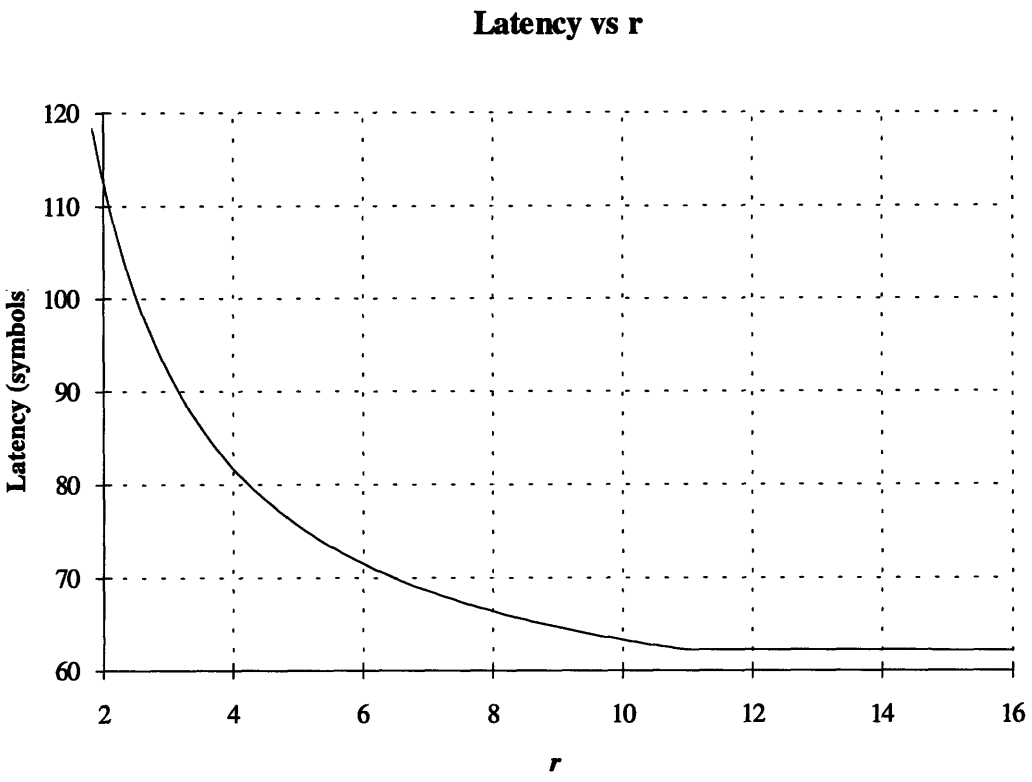


Figure 6-4 Latency as a function of variable processing rate

6.2.2 Required Buffering

As pipelined designs, both the original receiver and variable rate receiver require memory elements between successive stages. These elements are implemented in a straightforward manner in the original design, since they are read from and written to at

the same rate. However, in a variable rate design, more complexity is introduced, as separate addresses must be maintained for fast reads and slow writes. Also, in the overtaking case, the reads themselves may slow from one rate to another.

For example, in the original design, most of the memory elements are implemented in a first in first out (FIFO) or last in first out (LIFO) manner. A FIFO RAM stores oversampled IF data during the one burst latency of symbol timing. A LIFO RAM stores downsampled phases during the half burst latency of carrier recovery convergence. A reorder RAM stores compensated symbols as they available during the second half of carrier recovery, making them available in the correct order during the next burst. Finally, another FIFO RAM stores the serialized bit stream during the one burst latency of CRC decoding.

Many of the memory elements in a variable rate design can be implemented in a similar manner, as long as r is small enough such that carrier recovery does not overtake symbol timing. In this case, only the storage between symbol timing and carrier recovery requires independent read and write addresses. Furthermore, while the reads from and the writes to that memory occur at different rates, they occur at constant rates. All subsequent memories operate uniformly at the faster rate. However, if r is large enough to allow carrier recovery, or even decoding to overtake symbol timing, the fast memory reads must slow down to the symbol rate. This implies that all of the memories would have to support two read rates.

In order to quantify the latency / complexity tradeoff, examine Figure 6-4. Note that there is a diminishing return as processing speed increases. In a polling strategy such

as that suggested by MAC layer considerations, the most significant changes in latency occur for values of r where the amount of latency crosses a burst boundary. Since the latency remains between 1 and 2 bursts for any r greater than 2, there is little gain in performance for the large gains in complexity associated with very fast processing. The highest r for which carrier recovery does *not* overtake symbol timing, is that which satisfies the equality, $51+\frac{60}{r}=62$, obtained from the end time of carrier recovery, T_{B3}, meeting the symbol timing constraint, T_{A5}. Therefore $r=\frac{60}{11}\cong 5.45$. For implementation purposes integer multiples of the symbol rate or desired, so for $r = 5$, the latency is 75.6 symbols, or 1 burst and 40.625 us. If MAC layer work determines that a non-polling strategy is desired, such that successive transmissions are not piggybacked, a latency close to one burst is required to meet the 25 us inter-burst gap constraint. If so, then the highest r for which carrier recovery overtakes symbol timing but decoding does not, is the discontinuity in the total latency expression at $51+\frac{123}{r}=62+\frac{3}{r}$. This yields $r=\frac{120}{11}\cong 10.9$. Again choosing an integer result, $r = 10$, the latency is 62.3 symbols, or 1 burst and 5.99 us. Choosing $r > 10$ leads to excessive complexity in implementation of the decoder, with little reduction in latency.

6.3 Reducing Synchronization Overhead

Given the latency reductions already stated, the overhead requirements must be reevaluated. Given a receiver with very low latency, on the order of one burst,

piggybacking may not be required. As shown in Figure 6-5, now only 3 bursts need be transmitted for 1 burst of information to be sent.

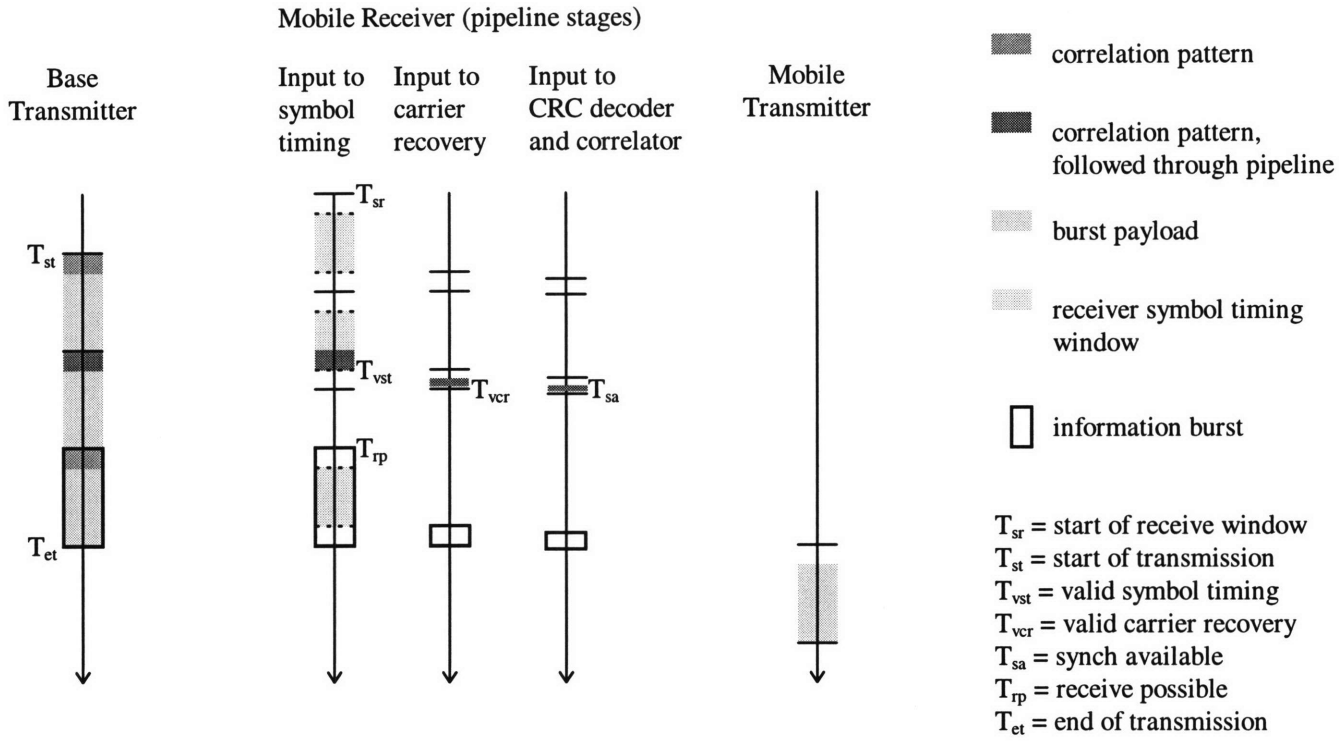


Figure 6-5 Overhead requirements of reduced latency receiver

6.3.1 Correlation Position in Burst

A simple method for reducing the overhead involved in synchronization, is to accept at least one burst of overhead and fill it with a series of synchronization words. As a result, the sliding window correlator will match on the earliest valid pattern, allowing the earliest possible synchronization. Each pattern would have to be distinct so that the correlator would be able to resynchronize accordingly. The benefit of this modification alone is unclear, since the correlation may occur after the second transmitted burst has already begun. In that case, synchronization would still occur at the start of the third

transmitted burst. The benefit of this method used together with others is studied further in simulation.

6.3.2 Parallelism - Multiple Staggered Receive Windows

Another method to reduce synchronization overhead, is to use a set of parallel demodulators with staggered symbol timing estimation windows. At one extreme is a set of ten demodulators *and* decoders, with windows staggered by 6 symbols each. Together this set could cover all positions of burst alignment that are within the slip correction range of each decoder. This configuration could entirely eliminate the need for one overhead burst. Further study is required to specify the minimum amount of parallelism required, but the case of 3 demodulators is instructive. This configuration, together with an initial burst filled with synch words, could reduce synchronization overhead to one burst.

6.3.3 Half Burst Demodulator

Finally, by shrinking the block size over which demodulation occurs, it becomes more likely that the first burst of a transmission will overlap a large fraction of the smaller symbol timing estimation window. For example, if the demodulator operated only on half bursts, with a smaller symbol timing window and half burst carrier recovery, the receiver would be more likely to resynchronize by the start of the second transmitted burst. The obvious consequence of this approach is worsened error performance, but simulation to determine the extent of performance loss is required. Also, analysis of the metrics used in symbol timing, and analysis of the carrier recovery loops, provides insight into the mechanisms behind performance degradation.

7. Analysis of Channel Estimates and Carrier Recovery

The existing demodulator treats each burst-sized block independently. This has the advantage of yielding averaged channel measurements well correlated to the channel at the measurement time, which is important in the case of a slowly time varying channel. Also, the short, burst-long measurement time allows a low latency implementation. However, for a non-varying channel, averaged measurements over a longer time period are more accurate, ostensibly permitting lower error rates. The price paid is susceptibility to time varying channels, where a single averaged measurement does not closely represent the channel at different times. Also, the longer measurement time requires a higher latency implementation. Thus, performance and latency tradeoffs are apparent in choosing the duration of the measurement time.

In addition, in examining receiver performance in asynchronous applications, the case where the measurement window does not fully overlap the time period of a valid transmission is of particular interest. In this case, the averaging measurements are degraded by the initial ambient noise of the channel, and are further degraded by the shortened period of available valid transmission. Furthermore, in the carrier recovery stage, the initial noise delays loop convergence on phase and frequency offset estimates, and shortens the length of valid input to drive the loop to convergence. As a result, insight into the sensitivity of channel measurements and carrier recovery to noise is required. This insight can clarify the performance tradeoff involved in the half burst demodulator suggested.

Finally, another consequence of a block oriented demodulator, is that symbol errors are necessarily incurred at block boundaries. Between blocks, information such as differential symbol encoding is lost as memory elements are cleared. Thus, while shorter window demodulators may be discussed for the purpose of synchronization, they are not appropriate afterwards, when burst-long CRC codewords must be preserved intact for successful decoding.

7.1 Symbol Timing Quality Index (QI) and Frequency Offset Estimation

As mentioned earlier, the PACS system uses square root raised cosine pulse shaping in transmitting the modulated stream. This pulse shape does have some inter-symbol interference (ISI), even when sampling at pulse maxima. However, a narrowband receive filter is employed, with a frequency domain characteristic resembling the square root raised cosine response. The cascade of these two characteristics ideally yields a raised cosine characteristic, leading to a no-ISI "Nyquist" pulse shape at the receiver. In the time domain, each raised cosine pulse is ideally infinite in time, however, each zero crossing of a given pulse coincides with the peaks of neighboring pulses. Thus, if the receiver chooses the correct sampling instant over a symbol period, there is no ISI. Figure 7-1 depicts a series of raised cosine pulses. Note that the pulse amplitude decays rapidly in the time domain. Thus, the pulse centered at $t = 0$, which is overlaid with contributions from the 2 pulses before and after, encounters interference typical for any given symbol.

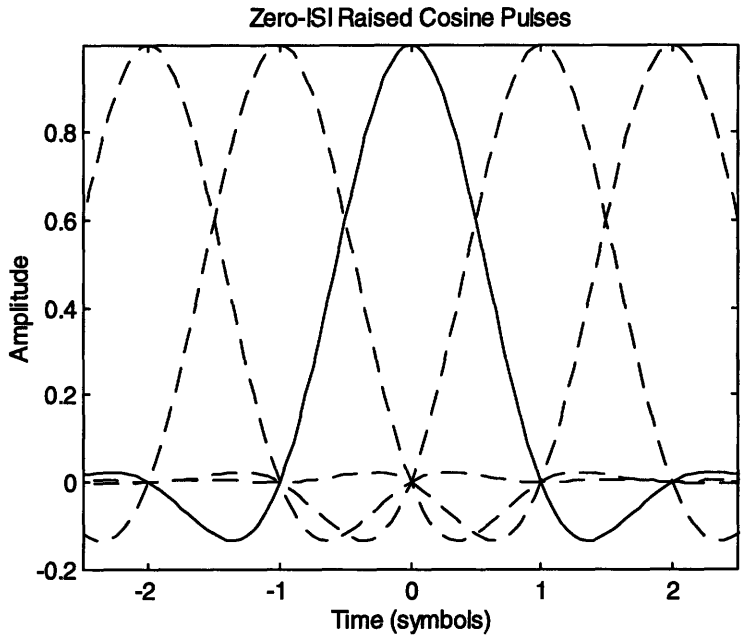


Figure 7-1 Zero ISI Pulse Shapes

It is the goal of symbol timing to estimate the correct sampling offset, updating the estimate each burst. This is done in the digital domain, in the following manner for each burst. An analog IF input is over-sampled with 4 bits of precision at a rate 20 times the symbol rate. Through a series of signal processing methods a quantized instantaneous absolute phase stream at 10 times the symbol rate is extracted from the over-sampled IF stream. Phases are represented with 8 bits of precision, proving 256 levels between $-\pi$ and π. At 10 offset positions this phase stream is differenced yielding 10 symbol rate differential phase streams. A QI metric is obtained for each sampling offset by operating on the corresponding differential phase stream, and the sampling offset with the highest QI is chosen for symbol timing. Figure 7-2 illustrates the band pass to phase stream conversion, the QI computation, and pipeline buffering to preserve the over-sampled

phase stream while a new decimation offset is chosen on the basis of QI. A frequency offset estimate and a symbol rate absolute phase stream are inputs to the next stage, carrier recovery. While this figure accurately represents the functionality of the implementation, the implementation is more hardware efficient.

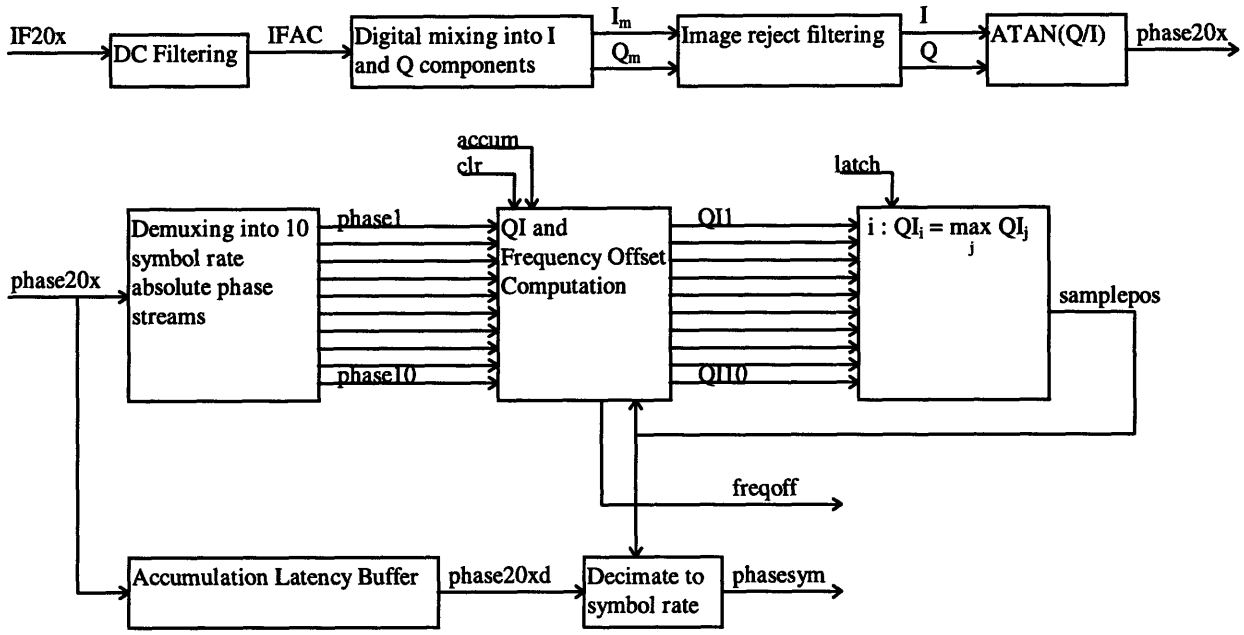


Figure 7-2 Bandpass to Phase Conversion

The QI metric is best understood as the magnitude squared of a vector sum of a series of unity magnitude vectors. Given an absolute phase stream, a differential phase stream is easily computed. In the absence of a frequency offset, when sampling at the correct sampling instant, the only acceptable values of differential phase should be clustered near the constellation points of +/- $\pi/4$ and +/- $3\pi/4$. Multiplying any of these acceptable differential phases by 4, yields π in all cases. The differential phase stream

multiplied by 4, $4\Delta\theta$, can be considered a stream of unity magnitude vectors. Noise, modeled as additive phase noise, may perturb each vector slightly from π. A vector is constructed by summing a series of the $4\Delta\theta$ vectors. Assuming zero mean noise, applying the law of large numbers, the noise should sum destructively while signal component sums constructively, yielding a large vector in the direction of π. The duration of the measurement time directly impacts the validity of the above assumptions, since over short periods of time a given noise sequence may not sum to near zero. Also, the amount of valid transmission within the measurement window affects the constructive summing of signal components. The less valid transmission that is accumulated, the closer the maximum QI will be to the smaller erroneous QIs. The less invalid transmission that is accumulated, the closer the erroneous QIs will be to the maximum QI. Figure 7-3 illustrates the computation of QI for a given symbol rate phase stream.[7]

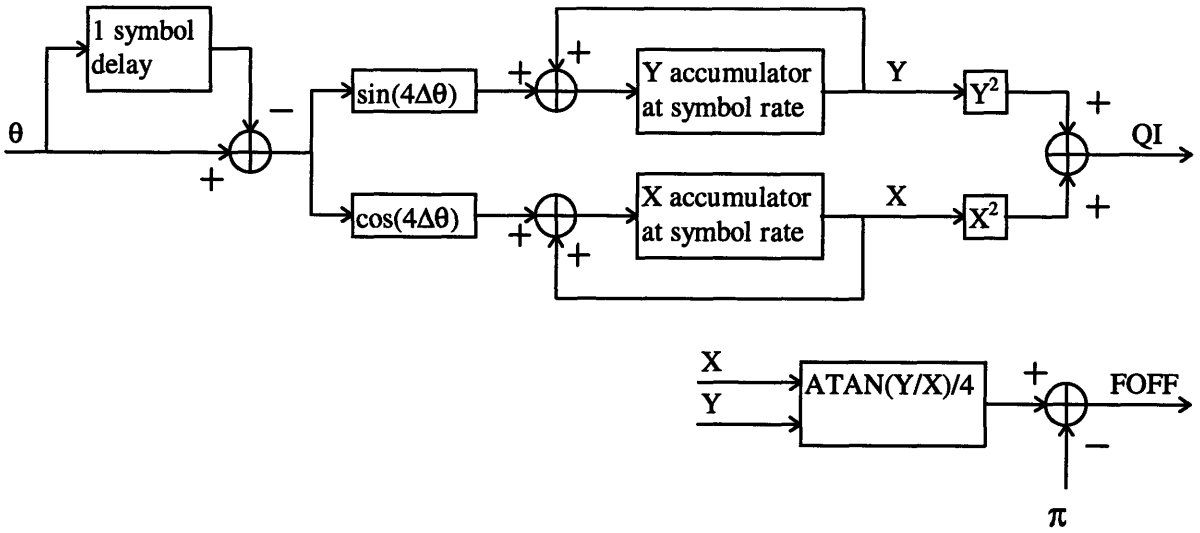


Figure 7-3 QI and Frequency Offset Computation

Sampling at a position offset from the correct one, ISI is encountered. It is no longer true that differential phases will be near constellation points. The effect of ISI can also be modeled as a data dependent noise, which increases as the offset from the correct sampling point increases, until one passes the center of the symbol and nears another sampling instant. Instead of performing this analysis, an empirical QI as a function of sampling offset, without noise, shows how ISI leads to a destructive vector sum away from the correct sampling point. Figure 7-4 was generated in simulation neglecting quantization, using an appropriately modulated, noise free, random bit sequence, with QI accumulated over approximately 960 symbols and normalized to a maximum value of 1. Assuming noise still sums destructively, a maximum QI can be used to choose the correct sampling instant.

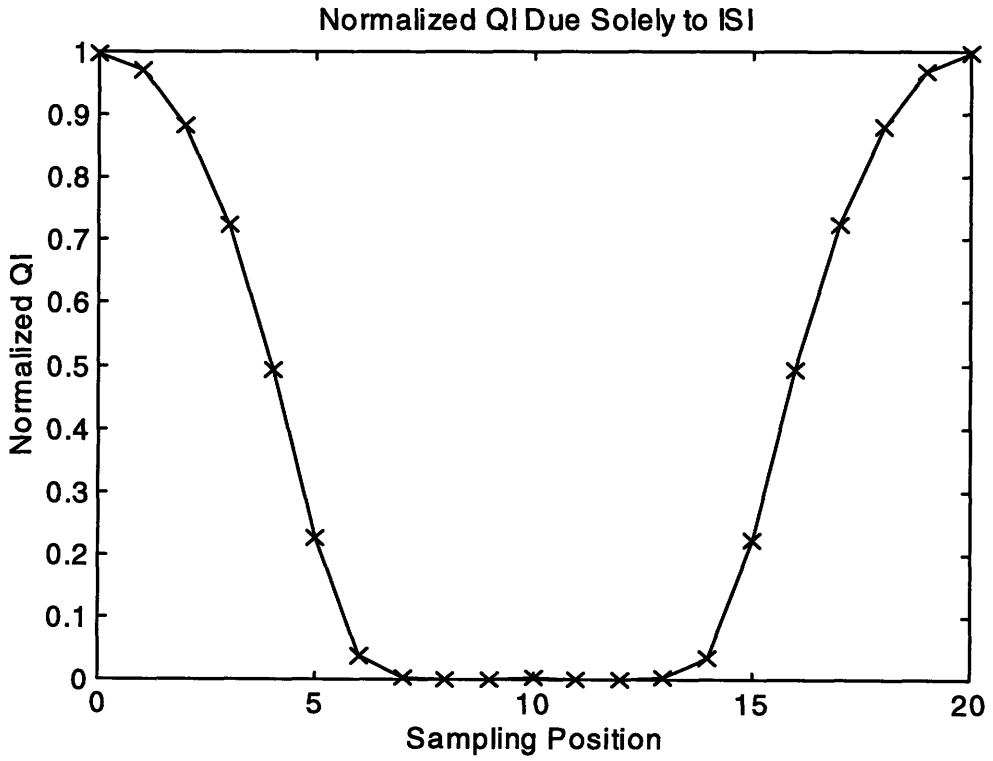


Figure 7-4 QI as a function of sampling offset in the absence of noise, (long accumulation).

In the presence of frequency offsets, the differential phase constellation is rotated by some angle δ. This is because a frequency offset can be represented as a constant derivative of phase, so in some time interval Δt there is a fixed phase change $\Delta\theta$. A given frequency offset then leads to a fixed phase change δ over each symbol interval, simply rotating the differential phase constellation by δ radians. Therefore, the magnitude properties of the QI vector sum without frequency offset, also apply in the presence of a frequency offset. However, with a frequency offset, the angle of the vector sum for the correct sampling instant will no longer be π radians. Instead, at the correct sampling instant, each vector of the $4\Delta\theta$ stream generally points toward $\pi + 4\delta$, and so will the vector sum. So while QI, the magnitude of the vector sum, indicates the correct sampling instant, the deviation of the angle of the vector sum from π indicates the frequency offset. The limitation of this method of frequency offset estimation is also apparent, since if 4δ exceeds π, a positive frequency offset would alias into a negative one and vice versa. This translates into a maximum frequency offset of +/- $\pi/4$ radians per symbol, which at a symbol rate of 384 kHz yields a frequency offset of +/- 48 kHz.

In order to investigate the effect of noise on the QI metric, several assumptions may be made. First quantization error will be neglected for initial ease of analysis. Second, the transmitted data sequence will be considered independent of the noise. As a result, the effect of ISI at sampling positions offset from the correct one will be treated as independent of the noise, since ISI is a deterministic function of the transmitted data. Furthermore, the effect of ISI may be considered a deterministic function of sampling offset conforming to the simulation results cited previously. While this is only directly

valid for very long accumulation periods with random data, it can be made valid by construction of appropriate short data sequences. Since user data will be preceded by some form of synchronization sequence, this sequence can be designed to yield ISI conforming to the assumption. Third, frequency offset will be considered a fixed differential phase offset within the estimation range, so that the case without frequency offset can be considered without loss of generality. Given these assumptions, one can derive some expressions for QI at the correct sampling point to see its functional dependence on noise terms.

$$
\begin{aligned}
\theta_i &= s_i + n_i + f_i \\
f_i &= 0 \\
\Delta\theta_i &= \theta_i - \theta_{i-1} = s_i + n_i - s_{i-1} - n_{i-1} \\
&= \Delta s_i + \Delta n_i \\
\Delta s_i &= (2k-1)\cdot\frac{\pi}{4} \\
QI &= \left\| \sum_i e^{j4\Delta\theta_i} \right\|^2 \\
&= \left(\sum_i \cos(\pi + 4\Delta n_i) \right)^2 + \left(\sum_i \sin(\pi + 4\Delta n_i) \right)^2 \\
&= \left(\sum_i \cos(4\Delta n_i) \right)^2 + \left(\sum_i \sin(4\Delta n_i) \right)^2
\end{aligned}
$$

This expression for QI demonstrates the effect of several types of phase noise. If $4\Delta n$ is uniformly distributed, then QI is very unhelpful, as it approaches zero. If it is gaussian distributed with zero mean, QI will approach N^2 in the absence of noise, where N is the number of symbols summed. The uniform distribution is a very pessimistic case, however. This may be justified with some empirically obtained cummulative density functions, that

reveal that white gaussian phase noise is a good approximation for a wide range of signal to noise ratios. First, one may generate a large number of independent trials of complex noise phasors with rayleigh distributed amplitude and uniform phase. For the empirical method, 100,000 trials were performed. Then, a signal component can be considered with unity magnitude and zero phase, or real part one and imaginary part zero. A resultant random variable can be computed for each trial that is the sum of the noise and signal components. The phase of this resultant for each trial is the phase domain noise of concern. An empirical CDF can be constructed based on the empirical distribution of the many trials, and this CDF can be compared to the empirical CDF for a gaussian with matching variance. If signal-to-noise ratio (SNR) is defined as $10\log_{10}(\text{var}(S)/\text{var}(N))$, and this is used to choose the rayleigh parameter a, the gaussian approximation can be tested at various SNRs. Figure 7-5 depicts the empirical CDFs obtained at two SNRs, and the difference between the rayleigh and gaussian ones. It becomes apparent that the approximation is least accurate in the tails of the distribution, or at the outlying values. However, the QI metric seems valid for the type of noise expected, at reasonable values of SNR.

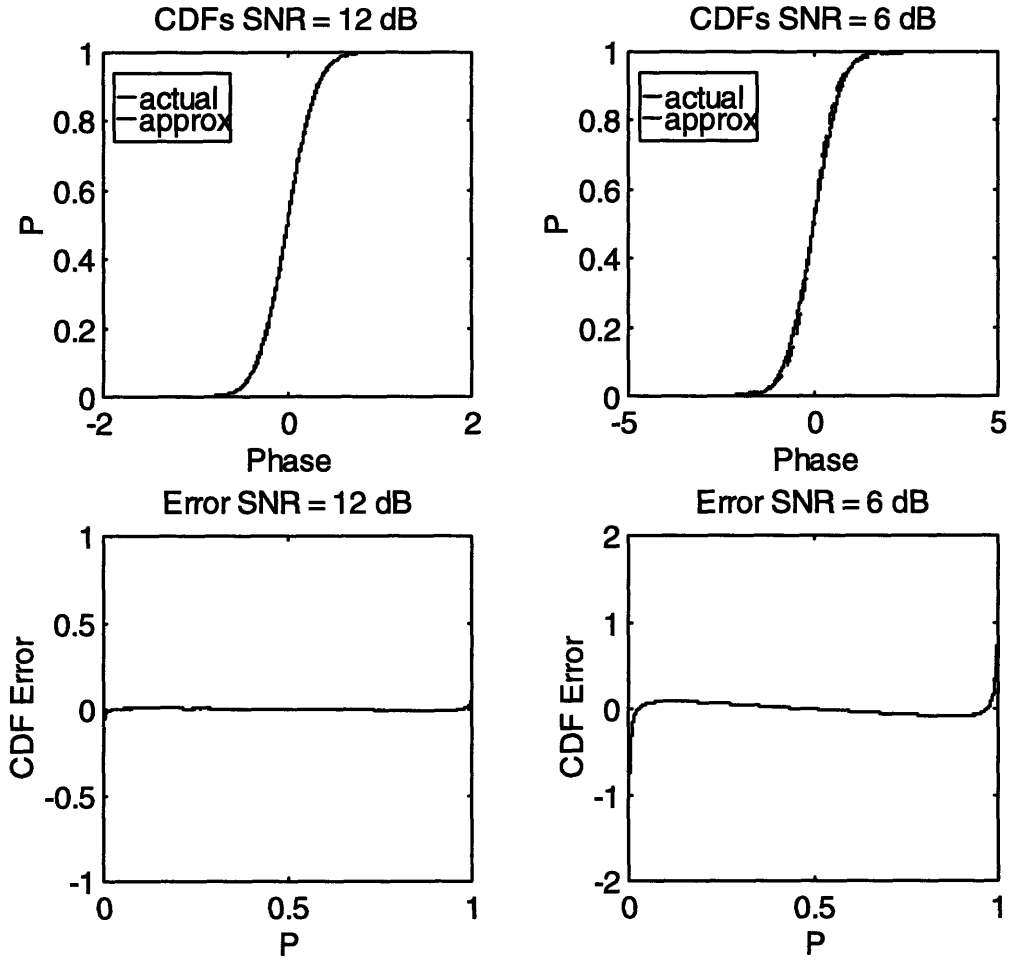


Figure 7-5 Empirical CDFs at several SNRs

7.2 Carrier Recovery Second Order Loop

The purpose of the carrier recovery stage is to refine the frequency offset estimate and converge on a phase offset estimate. The stage then compensates the symbol rate absolute phase stream for these offsets, extracting the symbol information as a sequence of di-bits. This is accomplished using a second order loop, with one state tracking phase offset, and a second state tracking frequency or change in phase offset. The frequency

offset state is initialized with the frequency offset estimate obtained in the symbol timing process, while the phase offset state is initialized to zero.

In the absence of frequency offsets, the correctly sampled, symbol rate absolute phase stream should ideally take values near the eight absolute phase constellation points. This would suggest that there are eight decision regions, however, there are actually only four. If one considers the absolute phase constellation at odd intervals of time and at even intervals of time separately, one obtains two constellations of four points offset by $\pi/4$ radians. This is depicted in Figure 4-1, with one sub-constellation denoted with boxes, and the other with circles. As a result, before the absolute phase stream is fed to carrier recovery, it is *derotated.* In other words, given the starting symbol phase in a sequence, $\pi/4$ radians is added to every other symbol phase, rotating one of the 4 point constellations to overlap the other. This derotated absolute phase stream, θ_d, has a simple 4 point constellation. The presence of frequency offsets simply causes the constellation to rotate from one symbol to the next.

The second order loop tracks phase and frequency offsets by driving phase error to zero. Phase error is defined as the phase difference between a given derotated symbol phase and the nearest constellation point. The loop is depicted in Figure 7-6. Again, the figure conveys the loop's functionality, but there are implementation specific details that are not shown. For example, the difference between $\theta_d(k)$ and the phase state register p(k) yielding error e(k), is only taken modulo one quadrant of phase, to consider error only from the nearest constellation point. As the figure shows, phase error is fed back with appropriately chosen gains a and b, to update phase and frequency offset estimates.

Loop analysis reveals that with the proper choice of a and b, e(k) will converge to zero. In the implementation it is assumed that the loop will converge within 30 symbols of iteration, with the input stream buffered during this convergence time. In the full burst receiver implementation, the loop continues forward through the remaining 30 symbols of the burst, while a functionally identical loop is run back through the buffered 30 symbols. The backward loop is initialized with the convergent state of the forward loop, reversing the sign on the frequency state for time reversed operation. This is the compensation stage, where the difference between $\theta_d(k)$ and the phase state p(k), is now considered over the full range of phase, not simply for error information. The resulting forward and backward phase streams should have a 4 point constellation on the axes of the complex plane. The constellation is rotated by $\pi/4$ radians by adding $\pi/4$ to each phase, and then the upper 2 bits of the resulting phases determine which of 4 symbol decision regions each phase is in. After some latency required for reordering this dibit stream to match its order of arrival, the reordered dibit stream is rerotated and differentially decoded to extract the symbol information.

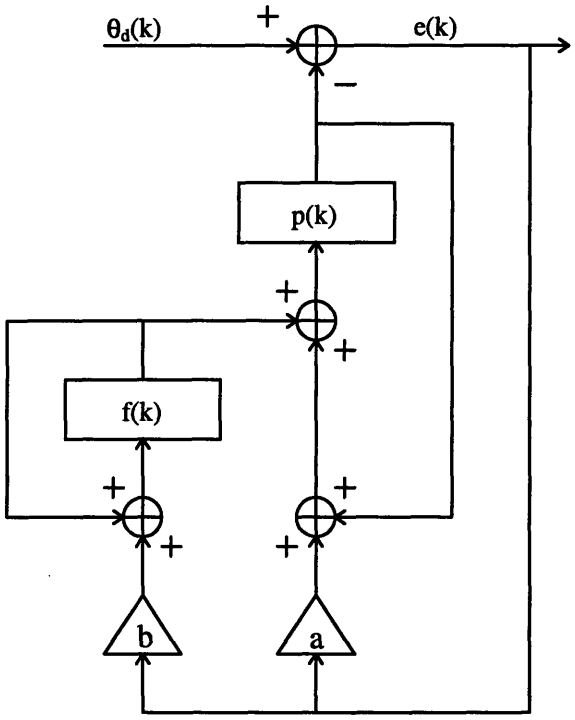


Figure 7-6 Carrier Recovery Second Order Loop

A key issue is the effect of noise on loop convergence. Also of issue is the case where the 30 symbol convergence window does not fully overlap a transmission. The presence of initial noise delaying convergence, coupled with a shorter information sequence for achieving convergence, raises the possibility that the loop will not converge. To study these issues, one may model the loop with a z-domain transfer function. Solving the difference equations obtained for the above loop, and expressing results in the D domain ($D = z^{-1}$):

$$\frac{E(D)}{\theta_d(D)} = \frac{(1-D)^2}{1+(a-2)\cdot D+(1+b-a)\cdot D^2}$$

By inspection the zeros are both located in a double zero at $z_z = 1$. Solving for the poles, one obtains:

$$D_p = \frac{2 - a \pm \sqrt{a^2 - 4b}}{2 \cdot (1 + b - a)}$$

In order to obtain a transfer function with desirable step response characteristics, such as short settling time and low overshoot or oscillation, a and b can be chosen such that the loop is critically damped. This constrains both poles to be a double real valued pole at $z_p = (2\text{-}a)/2$, with b equal to $a^2/4$. For a stable response, the poles must be within the unit circle, so $a > 0$. This is consistent with a negative feedback configuration. Furthermore, for greater immunity to noise and oscillation, $a < 1$. For ease of implementation, a and b are chosen as fractional powers of 2, 2^{-k}, since these multiplications can be represented with simple binary right shifts. Finally, the larger a and b are, the closer the poles are to the origin of the z-plane, and the faster the loop will converge. Based on these tradeoffs, a = ½ and b = 1/16, placing the double pole at z = ¾. Figure 7-7 shows the pole and zero locations chosen for the carrier recovery loop.

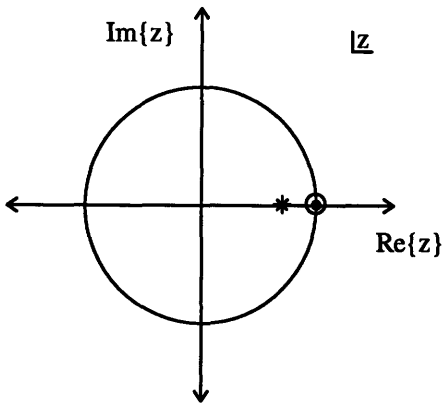


Figure 7-7 Pole and Zero Locations for Carrier Recovery Loop

In the absence of noise, this loop can track step inputs or ramp inputs, with error converging to zero within 30 iterations. Step inputs correspond to a derotated phase stream with a constant phase offset, while ramp inputs correspond to a derotated phase stream with a constant frequency offset. Figure 7-8 demonstrates error convergence for step and ramp inputs with slightly perturbed initial conditions, for both the chosen fast poles and alternate slower poles.

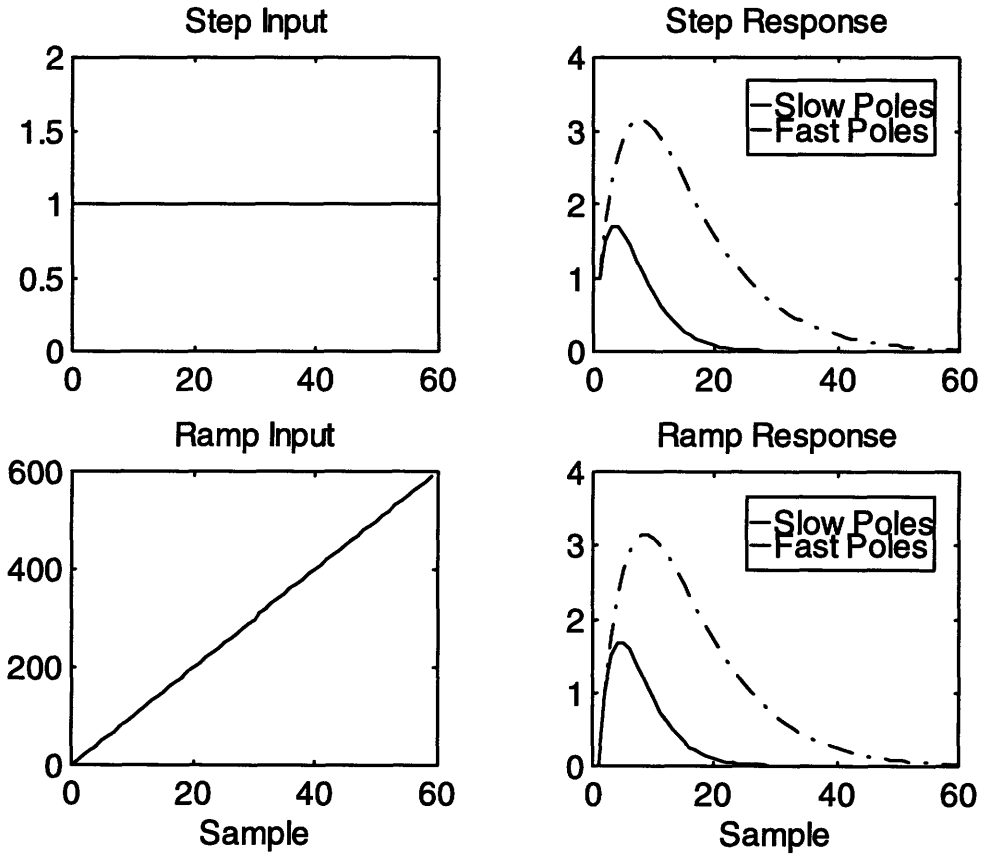


Figure 7-8 Step and Ramp responses for carrier recovery loop

The final value behavior of the loop can also be understood from a frequency domain perspective. The loop transfer function has a notch characteristic in magnitude. Both the step and ramp inputs have most of their spectral energy near dc, as a result, the cascade of these systems with the loop leads to a zero output.

The effect of noise on loop convergence must be considered in two steps. First, when the receive window does not fully overlap the transmission, the initial input to the loop is noise. Since the loop has state including only the last two inputs at any given time, the initial noise can be considered solely as perturbing the loop's initial conditions, before the start of valid data. Given valid symbol timing, the loop's frequency state is initialized to a value close to its final one, and the phase state is arbitrarily initialized to zero. The effect of initial noise is to perturb these states. Second, additive noise on the valid derotated phase stream may affect loop settling time and oscillation. By linearity the loop output can be considered a sum of responses to individual loop inputs. The above analysis shows that the output due to signal component converges to zero, so the output due to the noise component can be considered independently. Given low feedback gains, a and b, after convergence the loop error closely matches the additive noise. Again a frequency domain perspective adds insight. A wide-band noise process has energy beyond low frequencies that passes unchanged through the loop filter, and only components near dc are removed from the noise. The lower the feedback gains, the closer the poles become to the unit circle, and the loop filter becomes a more selective notch, removing components only very near dc. Important questions then, are what input noise is expected, and what is the critical noise energy that leads to unacceptable error oscillation. Unacceptable

oscillation has amplitude approaching $\pi/4$ radians, or an envelope of $\pi/2$ radians, since error spikes of this magnitude lead to aliasing of one symbol for another.

This analysis of carrier recovery provides insight into a "critical" offset, the time offset between the demodulation window and the transmission after which demodulation is no longer reliable. As the response plots show, at least 20 symbols of valid input are required for the loop to converge, corresponding to a critical offset of 10 symbols. While the demodulator may succeed under offsets greater than 10 symbols, it is not likely to succeed under offsets greater than 15 symbols. In this regime, even if symbol timing is successful, carrier recovery is likely to be highly divergent.

8. Whole Burst Receiver With Backward Buffering

One approach in modifying the receiver is to accept the synchronization overhead, but buffer the lost transmission, so that after synchronization, the receiver can simply run through the buffered data. Using this approach, the transmission that is used during synchronization is not wasted, and its information can be recovered after the fact. The disadvantage of this approach is the large amount of buffering required. Another disadvantage is that the terminal may waste time at the end of a transmission processing what remains in its buffer. This is the case unless the terminal can begin responding as soon as the base completes its transmission, while the terminal is still processing buffered data. Finally, this approach imposes a limitation on the minimum transmission size. The transmission must be at least as long as the long synchronization time, otherwise the terminal will be unable to synchronize, and any buffering is wasted. Actually one may

require a slightly longer transmission to allow the terminal time margin for initiating a response.

The large amount of buffering required renders this approach undesirable. Since the start of the transmission is not known initially, the buffer must store data at the over sampled rate, since the block demodulator would induce errors at window boundaries. Storing approximately 4 bursts or 2400 samples of 8 bit phase data would require an additional 18.75k of memory. Therefore, this approach alone is inefficient. However, if combined with the reduced latency implementation, only 2 bursts or 1200 8-bit samples, totaling 9.375k need be stored. In general, this approach becomes more desirable if combined with a reduced synchronization time receiver, reducing the amount of buffering required.

9. Half Burst Receiver

A more hardware efficient approach for reducing synchronization time is to employ a shorter-window receiver. The primary benefit to this approach is that a shorter measurement period allows a higher measurement frequency. If the start of the transmission is greatly offset from the start of a given measurement window, the first partially overlapping measurement will be unreliable. However, since the window is short, the next fully overlapping measurement occurs and ends soon enough after to allow rapid synchronization. As discussed earlier, the disadvantage of this approach is less reliable symbol timing and frequency offset estimates, and estimates and carrier recovery more sensitive to noise. The tradeoff then is faster synchronization time for higher probability

of missing a transmission. Two important questions then, are how fast can synchronization be, and how much does the miss probability increase.

A convenient choice for short window length is half of a burst, since that would permit an implementation of both a half and a whole burst receiver with a common data path, but with different control paths. This common architecture is important because after synchronization, a whole burst receiver is more desirable. The half burst receiver would introduce errors between half burst blocks at the window borders. Also, the whole burst receiver would mostly likely have better symbol timing and frequency offset estimates from the longer measurement time, given that the channel is flat over a given burst.

Even with a half burst receiver in the pre-synchronization stage, multiple synchronization words must be transmitted to permit rapid synchronization. If the first partially overlapping measurement window fails to provide valid symbol timing, even if the next window yields valid symbol timing, there must be another symbol sequence available to synchronize to. While there are highly refined methods for choosing the optimal length and values of synchronization sequences, the PACS frame synchronization sequence may be considered as a starting point. The PACS frame-sync sequence is 24 bits, or 12 symbols long. One could adapt this sequence to choose a set of several synchronization sequences, starting the transmission with a series of synchronization words. The receiver may then employ a different sliding window correlator for each expected sequence, determining the offset of its measurement window based on which correlator obtains a match. Following a correlation match, the receiver can realign its window and begin full

burst operation. From the above discussion it becomes apparent that the series of synchronization words should be chosen to differ as much as possible, having a large minimum distance, to prevent the possibility of errors aliasing one valid synch word to another.

In order to determine the synchronization capabilities of a half burst receiver, it is necessary to revisit the latency analysis and implementation issues enabling a common half / full burst architecture. The only activities necessary before synchronization are demodulation and correlation, since error checking is not possible until the block location is determined. As discussed earlier, demodulation occurs in two stages, a short 30-symbol symbol timing measurement, followed by pipelined carrier recovery on the symbol rate absolute phase stream. The resulting symbol stream is converted to a bit stream and correlated with several candidate patterns. The symbol timing stage lasts half a burst, and the following carrier recovery stage can begin only after this stage ends. However, the half burst carrier recovery can then be run at any rate, r, achievable in implementation, since all input required is available at the start of the procedure. Furthermore, the backward loop cannot begin until after the forward loop ends, hopefully converging. Thus the carrier recovery stage lasts for $60/r$ symbols in time, half of the time in forward operation and half backwards. The correlation operation cannot begin until after the end of carrier recovery's backward loop, when the first symbol of the half burst window is finally recovered. However, the correlators may then be run at any achievable rate. Given that the earliest available sync sequence within the half burst window ends at symbol n, correlation then lasts n/r symbols in time. See Figure 9-1 for the latency analysis in the case where $r = 5$. Note that the above procedure makes use of the same loops and symbol

timing data paths as the full burst receiver, the only difference is the duration of symbol timing measurement accumulation, and the lack of simultaneous forward / backward carrier recovery.

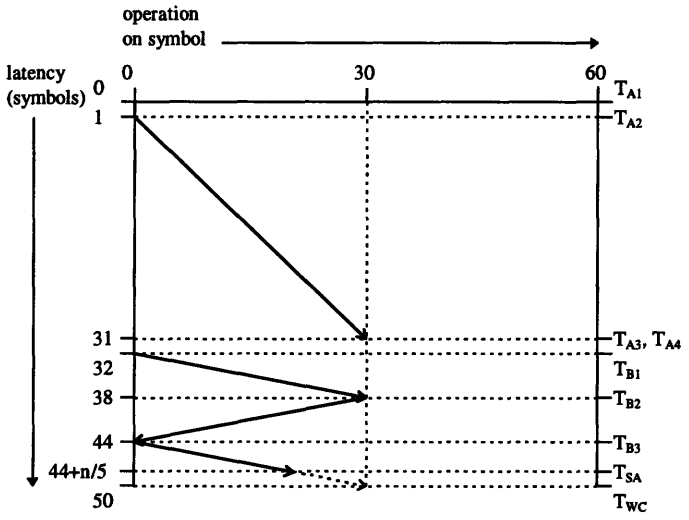


Figure 9-1 Latency of Half Burst Receiver

It is possible to synchronize by the beginning of the second transmitted burst, depending on the amount of overlap required to obtain valid symbol timing and carrier recovery. Simulation has shown that offsets up to one third of the measurement window can still yield valid symbol timing. After the offset exceeds one half of the window, the following fully overlapping window is likely to provide valid symbol timing, carrier recovery and synchronization before the beginning of the second transmitted burst. For the difficult case of offsets in between these values, if the partially overlapping window fails to provide valid symbol timing, the next fully overlapping window must be relied on. In this case, in order to synchronize by the start of the second burst, the carrier recovery operation must be performed at a very high rate. Thus, the smallest offset at which

symbol timing first fails will constrain the amount of time between the end of the second window and the beginning of the second burst, the time in which carrier recovery must be completed. This then constrains the lowest rate at which carrier recovery can be performed, still guaranteeing synchronization by the second burst. Simulation suggests that a rate between 8 and 10 times the symbol rate is required. See Figure 9-2 for an illustration of this timing, for the case where $r = 10$.

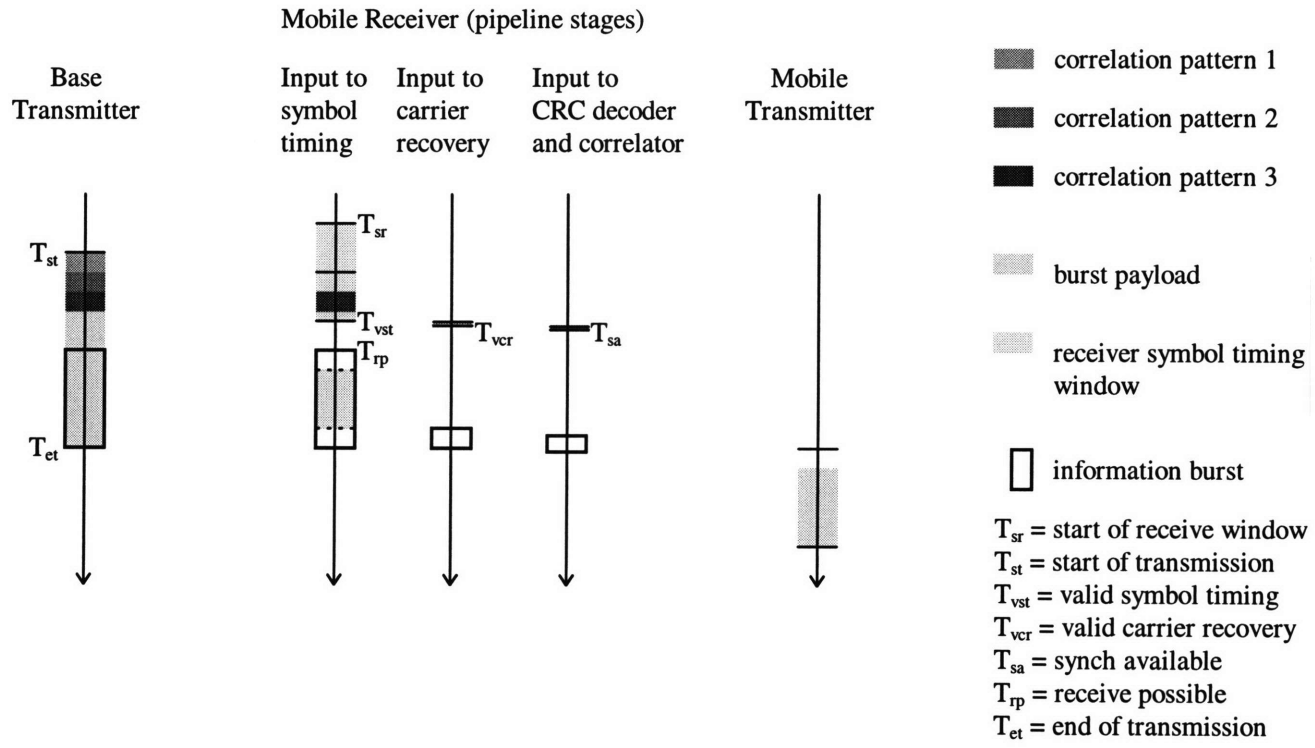


Figure 9-2 Synchronization Timing for Half Burst Receiver

Note that the partially offset measurement window is sufficiently offset that symbol timing or carrier recovery is invalid. Also note that after synchronization, the receiver switches to variable rate full burst operation. The variable rate operation is still required to allow the terminal to respond shortly after the end of the base's transmission.

If the receiver is designed to ensure synchronization by the beginning of the second burst, it becomes apparent that no more than three 12 symbol synchronization sequences, 36 symbols, are needed. Synchronization to any sequence after these three would occur so late in the burst that it would not be in time for the beginning of the second burst. As a result, a limited backward buffering approach could be employed in which only 24 symbols, or 240 8-bit phase samples, need be buffered. Since 36 of the first burst's 60 symbols are used for synchronization, the remaining 24 can be used as the start of user data. As with any buffering scheme, this assumes that the terminal can begin responding after the base ceases transmission, while its receiver is still emptying out the buffer. In this scenario, the base wastes only 3/5 of a burst in its transmission. Without buffering the base wastes only 1 burst.

In order to determine the associated increase in miss probability, further simulation is required.

10. Highly Parallel Receiver

The above synchronization methods are all susceptible to the initial offset between the receiver window and the actual start of transmission. These methods tolerate an invalid partially overlapping measurement period, by accepting later synchronization. A receiver that employs greater hardware parallelism, may not be susceptible to the offset constraint. By using several parallel receive paths, with staggered receive window timing, a transmission that is offset from one receive window, may be aligned with another staggered one. From this discussion it seems that a sliding window demodulator would offer greatest flexibility, performing coherent demodulation over a measurement window

that slides through the received stream, without requiring the added hardware complexity of several parallel receivers. However, such a demodulator lacks clear symbol timing decision criteria to distinguish between local, absolute and spurious maxima of the QI metric. A staggered parallel implementation can allow each path to run independently, with the earliest correlation hit from any given path used to resynchronize the receiver. While this implementation may be more susceptible to detection false alarm, it would be less susceptible to detection misses.

Clearly a parallel implementation requires additional, undesirable hardware complexity. However, in choosing the degree of parallelism, one may trade between complexity and performance. At one extreme is a design with 10 parallel full burst receive paths that can synchronize to the start of the first transmitted burst with very limited overhead. This design takes advantage of the CRC decoder's ability to recover the transmitted codeword within a slip range of 6 symbols. The receive window of each successive path is staggered by 6 symbols. Together, the slip range of the 10 decoders spans the entire burst. However, the design also requires 10 parallel full burst demodulators, since the block orientation of the demodulator incurs errors at window boundaries, possibly within a codeword if the window is not synchronized. Synchronization to the first transmitted burst is possible after the CRC decode stage, not at the correlation stage. With the high degree of parallelism, one of the 10 decoders should yield a correctly decoded word, and knowledge of the path which does, together with the decode slip estimate, yields synchronization information.

While this approach provides high performance, it requires very high complexity. Instead a design of only three parallel receive paths should be considered. With such a design, synchronization is possible only at the correlation stage, not the decode stage, since there are gaps between the decoder slip recovery regions of the 3 paths. Thus, only rapid correlation is of concern. Full burst demodulation will suffer from the same drawbacks as when only a single receive path is employed. If a given partially overlapping window does not provide a valid demodulated stream, the next staggered window will yield results too late, after the end of the first transmitted burst. This conclusion results from the latency analysis of the full burst demodulator, which even with variable rate processing, has latency greater than 1 burst. The latency only up to correlation is bounded by 1 burst and 2 symbols, and a few additional symbols of time are required for the sliding window correlator to reach the correlation position in the demodulated bit stream. With only 3 receive paths, consider a successive window offset of 1/3 burst, 20 symbols. Staggering with any smaller offset defeats the parallelism, approaching the limiting case of a single receive path. With this staggering, there is a high likelihood that the partially overlapping window will not yield valid demodulation, for an offset from the transmitted burst of between 15 and 20 symbols. In this range, a majority of the 30 symbol input to forward loop carrier recovery is noise, allowing insufficient time for loop convergence. The next fully overlapping window from a staggered path, would end between at least 7 and 2 symbols later than the end of the first transmitted burst. Thus, if rapid correlation is desired, a half burst demodulator retains its advantage.

A set of 3 parallel receive paths, with staggered receive windows, employing half burst demodulators, has several advantages over a single half burst demodulator. The

optimal staggering is now 1/3 of one half burst, or 10 symbols. This design allows two possible benefits. First, given the suggested transmit structure of 3 12-symbol synchronization words, one may achieve a lower detection miss probability. The effective window repetition frequency has gone from 1 window starting every 30 symbols to 1 window starting every 10 symbols. Second, given a higher miss probability, one may achieve a lower overhead ratio, by providing fewer synchronization words in the transmission. Since a new window begins every 10 symbols, there is a high likelihood that synchronization will be possible before the conclusion of the second window in the first partially overlapping receive path. It is the fallback position of the second window, and a third synchronization word, that allows synchronization with a single half burst receiver. Two synchronization words may be sufficient if 3 parallel receivers are employed. In this case, combined with some backward buffering, overhead may be reduced to 2/5 of a burst.

A final advantage of parallel receive paths is possible after synchronization. Upon synchronization, the parallel receivers must switch to full burst demodulation, and are all realigned to the transmitted burst position. Now the parallel paths can be used to implement antenna diversity, choosing among as many antennas as paths are available. Receive path selection can be performed on an aggregate basis of highest QI and successful CRC decode. Studies by Bellcore have shown significant performance improvements through the use of antenna diversity in multipath environments[8].

11. Simulation and Evaluation

In order to evaluate the ideas presented above, several models have been constructed using MATLAB to represent typical modulated transmissions with various channel impairments, as well as several candidate receivers.

The transmissions are represented in the form ultimately presented to the digital logic receivers, abstracting out specific operations in the analog front ends. Starting with a source bit sequence, it is divided into appropriately sized blocks and CRC bits are calculated and appended to each block, according to the PACS 105,90 code. This bit sequence is converted into a symbol sequence, by mapping each 2-bit pair to a corresponding symbol. The symbol sequence is modulated into a symbol rate, complex phasor, with phase corresponding to $\pi/4$ shifted QPSK. This stream is upsampled to a complex impulse train at 20 times the symbol rate, which is then convolved with a finite time representation of the raised cosine pulse shape. This complex baseband stream is mixed up to an intermediate frequency of 820 kHz. Finally, the real part of the complex sequence is taken to provide a cosine at the chosen IF frequency with appropriate phase modulation. This sequence is then appropriately scaled and quantized so that its envelope occupies the full scale of the 4 bits available. It is this 4-bit, 20 times symbol rate, IF stream that is input to the candidate receiver. The modeling of channel impairments introduces frequency offsets and noise at several steps in this process, discussed in greater detail later.

The various candidate receivers are all implemented as variations on a model of the existing receiver. The basic model operates on burst-sized blocks of IF input, using a loop

to cycle through successive blocks. Each run of the loop includes an IF to phase conversion stage, a joint symbol timing and frequency offset estimation stage, a carrier recovery stage, a correlator and in some cases a CRC decode stage. The stages are implemented in a way functionally equivalent to the methods described in section 7, but they closely match the methods used in actual logic implementations. However, the sliding window correlator must operate across block boundaries. During each run of the loop a correlation bit stream is constructed by prepending the trailing portion of the previous block's output bit stream to the current block's output bit stream. A match is declared at the end of the first occurrence of the desired pattern in the aggregate bit stream. Simulated quantities are quantized appropriately and operations are represented with the same look up tables used in hardware. Also, the CRC decode stage actually consists of seven parallel CRC decoders, operating on time-staggered block locations, enabling the 6 symbol slip range discussed earlier. The various stages are written using vector operations where possible, to take advantage of MATLAB capabilities. As a result, there is no intrinsic concept of the processing latency involved in a corresponding hardware implementation. The manner in which latency is introduced to the model is discussed further below.

11.1 Front End Assumptions

Several details of front end operations have been neglected. First, assuming a raised cosine response is a simplification, since transmit shaping is square root raised cosine, and receive filtering is not a perfectly matched square root raised cosine. Also, the effect of pre-filtering and post-filtering on additive noise is ignored. Second, assuming that the received IF stream is not clipped and occupies the full scale of quantization is

another simplification. The true receiver may employ some form of limiting, and the gain of receive amplifiers and biasing of analog-to-digital converters may vary. Finally, basic design parameters of clocking and sampling are changed by using the new 384 kHz symbol rate. These parameters depend on the frequency of the analog IF input chosen, which leads to very different implementations of the bandpass-to-phase conversion stage.

With the previous 192 kHz sampling rate, a convenient solution for IF analog-to-digital conversion was possible. In early designs, the mixing of the sampled IF signal into I and Q components was done in the digital domain. This was possible, since the IF signal was sampled at 4 times the IF frequency, and sine and cosine waveforms could be represented as simple [1,0,-1,0] sequences. However, for symbol timing purposes, 16 samples per symbol were required, constraining the sampling rate to 16 times the symbol rate, and the IF frequency to 4 times the symbol rate, or 768 kHz. The design in simulation represents a newer technique that allows elimination of an IF conversion stage. A more common IF frequency of 10.7 MHz is used, and the input is band pass sampled below the Nyquist rate, at 3.84 MHz. At this rate, 20 samples per symbol are available. Spectrum information is not aliased, since it only occupies a band of 288 kHz, and the effective IF frequency becomes 10.7 MHz - 3*3.84 MHz, or -820 kHz. This can also be considered an IF frequency of 820 kHz with a phase inversion. Note that this is not one-fourth of the sampling rate, which is actually 960 kHz. This bulk 140 kHz offset is assumed and compensated by default, by subtracting an accumulated phase change from the calculated phase stream on a sample by sample basis.

Now, if the symbol rate is doubled to 384 kHz in order to occupy a band of 576 kHz, this architecture must be replaced. It may no longer be appropriate to employ band pass sampling to permit a convenient choice of IF frequency. If so, a design following the early implementation with IF frequency at 4 times the symbol rate, and only 16 samples per symbol, will have similar performance. However, for the purpose of simulation, the 192 kHz symbol rate is considered using the band pass sampling technique, together with bulk frequency offset compensation. This is acceptable since the bandwidth doubling is expected to effect front end implementation more significantly than it will effect baseband receiver performance.

11.2 Modeling Channel Impairment

Channel impairment that corrupts the modulated transmission occurs in several forms. First, relative differences in velocity between a mobile terminal and a stationary base can result in doppler shifts of the transmission carrier frequency. Second, in a multipath propagation environment, Rayleigh fading may occur. In this case the transmitted signal may be reflected off of various objects in the environment, arriving at the receiver via several paths of different lengths, and thus different phase delays. This can be modeled probabilistically as the sum of many complex phasors. Applying central limit theorem arguments to the resultant sum at the receiver, one obtains Gaussian distributions for real and imaginary parts. This assumption yields a carrier with Rayleigh distributed amplitude and uniformly distributed phase. Finally, thermal noise in the environment,

modeled as a white gaussian process is always present. This noise is significant, as it is the input to the receiver in the absence of a valid transmission.

For the purpose of simulation, carrier offsets can be represented by mixing the baseband signal to the carrier frequency plus a corresponding offset. It is assumed that only offsets within the compensation range of +/- 24 kHz, for the symbol rate of 192 kHz, are experienced. Note that the compensation range is doubled when the symbol rate is doubled. Also, for each test, the carrier offset is held fixed over the duration of the test, from one burst to the next. In modeling the multipath environment, appropriately scaled WGN is added independently to real and imaginary parts of the baseband complex phasor. Finally, the portion of input preceding the transmission in the partially overlapping window is modeled as WGN prepended to the IF sequence.

11.3 Modeling Quantization Error

Quantization error is a source of performance degradation that must not be neglected. As mentioned earlier, the receiver model accurately represents the quantization levels used in implementation. For example, the input IF is represented with 4 bits of precision, and various phase streams are represented with 8 bits. Functions such as sine, cosine, arctangent and square, are all represented with look up tables, with appropriately quantized outputs. The phase and frequency states of the carrier recovery loops are 14 bit registers. The frequency estimate is appropriately left shifted to initialize the frequency register. Loop gains are implemented with a series of right shifts. Only the 8 most significant bits of the phase state are used in subtraction from the derotated phase stream. Also, quantities are represented in 2's complement format when it is significant for certain

boolean logic operations. Thus, the receiver model matches the quantization errors introduced in a hardware implementation.

11.4 Modeling Implementation Latency and Synchronization

As discussed in section 6, latency is significant only in very loose terms. If the receiver acquires synchronization between one burst and the next, it cannot act synchronously with a burst start until the following burst. For this reason latency is simulated with very low granularity, only to the nearest burst or half burst. This is done by introducing delay variables in the receiver's block loop. Since the receiver loops through block sized portions of the IF stream, one can assume that the loop has a duration of one block. Latencies that are integer multiples of one block can then be represented by assigning outputs of one stage to a delay variable, whose output is in turn, assigned to the input of the following stage. Latencies that are less than one block can be aggregated into a net delay of one block. In the case of the half burst demodulator, a finer representation of latency is possible, as the loop duration is half a burst. In the case where variable rate processing is employed, such as in the carrier recovery loop of the half burst receiver, it can be assumed that carrier recovery and correlation is completed by the end of the half burst loop. With three parallel half burst demodulators staggered by 10 symbols each, three parallel demodulators are run during each loop iteration, processing sections of the IF stream that are appropriately offset. Latency in each parallel path is modeled identically, with latency of the different paths distinguished by the bulk time offset of 10 or 20 symbols between them.

Synchronization is modeled by translating a pointer used by the receiver to index the start of a block in the IF stream. This pointer is referred to as the processing pointer. For the full burst receiver, the processing pointer initially is set to the start of the IF sequence, and is incremented by one burst worth of samples (1200) at the end of each loop iteration. Upon the first correlation match obtained, the index of the start of the transmission is estimated using the index of the match and correcting for processing latency. This pointer is referred to as the start pointer. The time position of the match event relative to the start event is estimated by adding the processing latency of the increased rate window correlator to the current processing pointer and then subtracting the start pointer. If this relative synchronization delay is less than one burst, the match occurs early enough to allow synchronization by the start of the second burst, and the processing pointer is reinitialized to the start pointer plus 1200 samples. Otherwise the processing pointer is reinitialized to the start pointer plus 2400 samples. This is to model the fact that the receiver cannot synchronize to a position earlier in time than the point at which the match occurs.

For the half burst receiver, the processing pointer is again initialized to the start of the IF sequence. However the processing pointer is incremented by a half burst worth of samples (600), at the end of each loop iteration. Correlation is more sophisticated since there are now 5 synchronization words in the transmitted burst, but as mentioned earlier it appears that only 3 are required. Upon the first correlation match from any of three separate correlators, the start pointer is calculated as described earlier, with a correction included based on which synchronization word is detected. A pointer to the match event is estimated by adding the processing latency of the increased rate carrier recovery *as well*

as the increased rate window correlator to the current processing pointer. The time between the match event and the start event, the synchronization delay, is obtained by subtracting the start pointer from the match pointer. The processing pointer is reinitialized as described earlier. After synchronization, the full burst receiver is invoked with the reinitialized processing pointer.

In the parallel staggered receiver case, three independent processing pointers are maintained, one for each path. The first is initialized to the start of the IF stream, the second to an index 200 samples (10 symbols) later, and the third to an index 400 samples (20 symbols) later. As mentioned earlier, in the parallel case only 2 synchronization words appear necessary. Upon the first correlation match from any of 6 correlators, 2 per path, the start pointer is calculated, including an additional staggering offset correction based on which path yields the match. The position of the match event relative to the start event is estimated to determine sync delay. Again the processing pointer is reinitialized to the nearest burst beginning after the match event, and the full burst receiver is invoked with the new processing pointer.

A valid detection can be declared in the full burst, half burst or parallel half burst case, if the first block received after synchronization passes CRC error check.

11.5 Measuring Performance

Four measurements to evaluate the different receiver implementations include Prob.(miss), Prob.(false alarm), Prob.(correctly decoding packet), and worst case overhead to information ratio. The probability of miss is the likelihood that the receiver decides a packet has not been transmitted when one has been, and the probability of false

alarm is the likelihood that the receiver decides a packet has been transmitted when one has not been. These metrics provide insight into the ability of the receiver to even detect transmissions. The probability of correctly decoding a packet is the likelihood that, given a packet has been transmitted, it is received by the receiver error free. This metric is valuable as a basic description of link quality, especially in terms of service as a bit pipe to higher layer protocols and applications. This metric is also important in analyzing the performance of higher layer error recovery and retransmission schemes. These metrics must be qualified, however, by the worst case overhead to information ratio, which takes into account the proportion of the transmission that is wasted as overhead in the worst case scenario. Together, these performance metrics should be used to evaluate the suggested proposals under a variety of channel impairments.

Unfortunately, collecting these metrics involves much simulation. For example, one must determine a miss probability for a given receiver design, at a given level of channel impairment, and at a given receive window offset. For a probability on the order of 10^{-5}, several orders of magnitude greater than 10^5 trials must be performed for reliable results. Each trial consists of generating a sampled IF sequence corresponding to a short transmission, since only synchronization is of concern. Furthermore, to determine worst case performance, this set of trials must be performed over a range of possible time offsets between the start of transmission and the start receive window. Offsets can be simulated by pre-pending the transmission IF sequence with varying amounts of noise sequence. The size of the offset range simulated must be at least the duration of the receive window, in other words only offsets modulo one window length are of concern. These simulations must then be performed over a range of channel impairments, varying the transmission

signal-to-noise ratio first, and the pre-transmission signal-to-noise ratio second. In each of these simulations, the time at which synchronization occurs relative to the start of transmission indicates the required synchronization overhead, and thus the worst case synchronization overhead. These simulations must be repeated for each suggested receiver design. In order to determine probability of false alarm, which would result due to accidental correlation between a demodulated bit stream and one of several candidate 24 bit synchronization words, an even larger number of simulations may be required. If a binary sequence is chosen by succession of fair Bernoulli trials, then the probability of the resulting 24 bit sequence matching a given correlation sequence is 2^{-24}, on the order of $6x10^{-8}$. Thus, probability of false alarm might be better determined through real time testing of hardware prototypes. Finally, probability of correctly decoding packets after synchronization is a metric that can be best obtained from testing of existing prototypes. After synchronization, all of the suggested receivers operate in full burst mode, and although they employ variable processing techniques to reduce latency, they employ algorithms identical to those used in PACS telephony receivers. Thus the probability of correctly decoding packets after synchronization can be separated from the problem of synchronization, and can be measured by collecting statistics from working PACS telephony prototypes. Actually, curves of bit error rate (BER) versus signal-to-noise ratio have already been determined for early PACS receivers in several studies.[9]

Although statistically significant results for Prob.(miss) cannot be obtained with limited simulation, a sense of achievable worst case synchronization delay *can* be obtained for the various designs suggested. This delay can be considered a measure of the overhead required for synchronization.

12. Results

12.1 Testing Plan

As already mentioned, probability of miss simulation will not be performed. However, in order to determine the worst case synchronization delay, several simulations are performed upon the half burst demodulator and the parallel staggered half burst demodulator. The synchronization capability of the full burst demodulator is not explored in simulation, since analysis easily reveals its more limited performance. The simulations are performed for several typical signal-to-noise ratios of interest. The impairments simulated include three transmit SNR, pre-transmit SNR pairs, (12 dB, 14 dB), (9 dB, 9 dB), and (9 dB, 5 dB). Only a single frequency offset is simulated, since receiver performance is fairly flat across frequency offsets.[10] However, a significant offset of 12 kHz is chosen to demonstrate that the carrier recovery loop functions properly. For each of these impairments, the receiver was simulated for window offsets ranging from zero to 27.5 symbols duration, in increments of 2.5 symbols duration. It is assumed that offsets greater than 30 symbols, the window length, need not be simulated. Offsets exceeding 30 symbols simply include one or more fully non overlapping receive windows, and this behavior can be explored with probability of false alarm measurement.

Examples of the modulated waveform and channel impairment are provided in several figures. The effect of the raised cosine pulse shape on the absolute phase constellation is shown in Figure 12-1. The constellation shows the complex phasor sequence resulting from a long random symbol sequence, to show as many of the possible trajectories as possible. In this example there is no frequency offset, and the start of the trajectory sequence from the origin has been omitted. Eight constellation points are visible

as the intersection of many phasor trajectories, but note that pulse shaping leads to a sequence of phasors in the transition from one constellation point to the next.

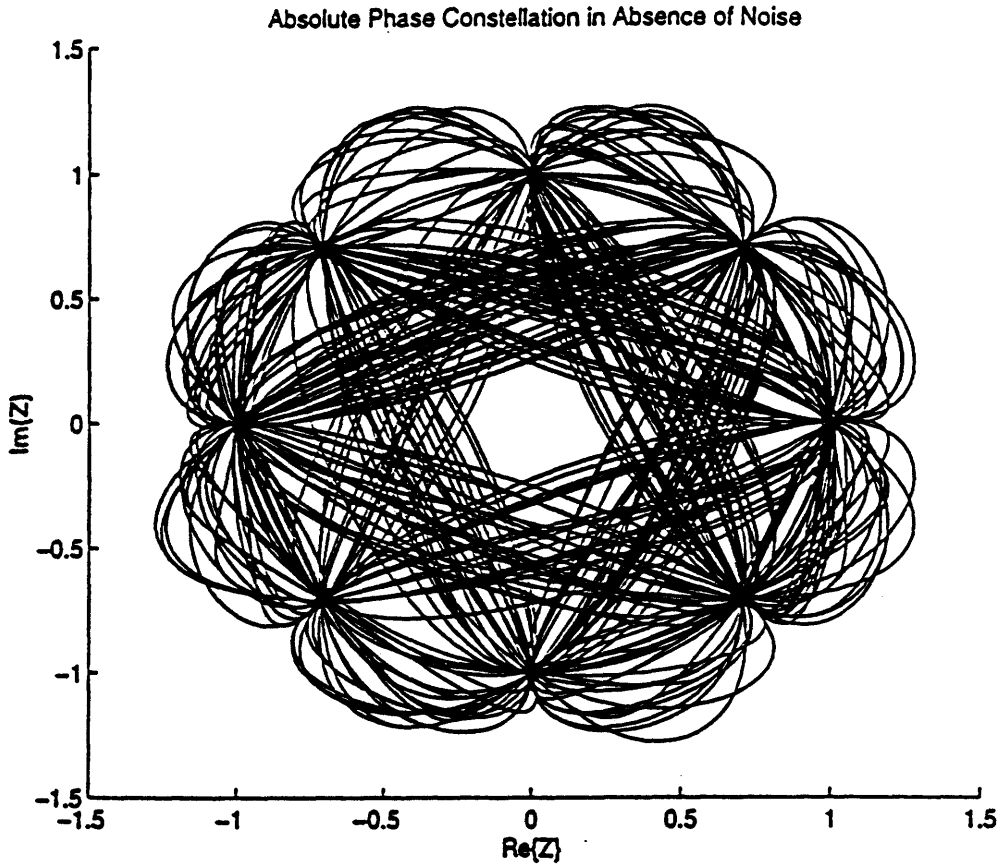


Figure 12-1 Pulse Shaped Absolute Phase Constellation

Three values of pre-transmission noise are displayed in the time domain waveforms of Figure 12-2. Each waveform is a sampled quantized input IF stream. In this example, the offset simulated is 200 samples or 10 symbols duration. Note that the modulated waveform does begin until sample 201. Thus, if the processing pointer is initialized to the

start of the IF stream, it has an effective offset of 10 symbols from the start of transmission.

.

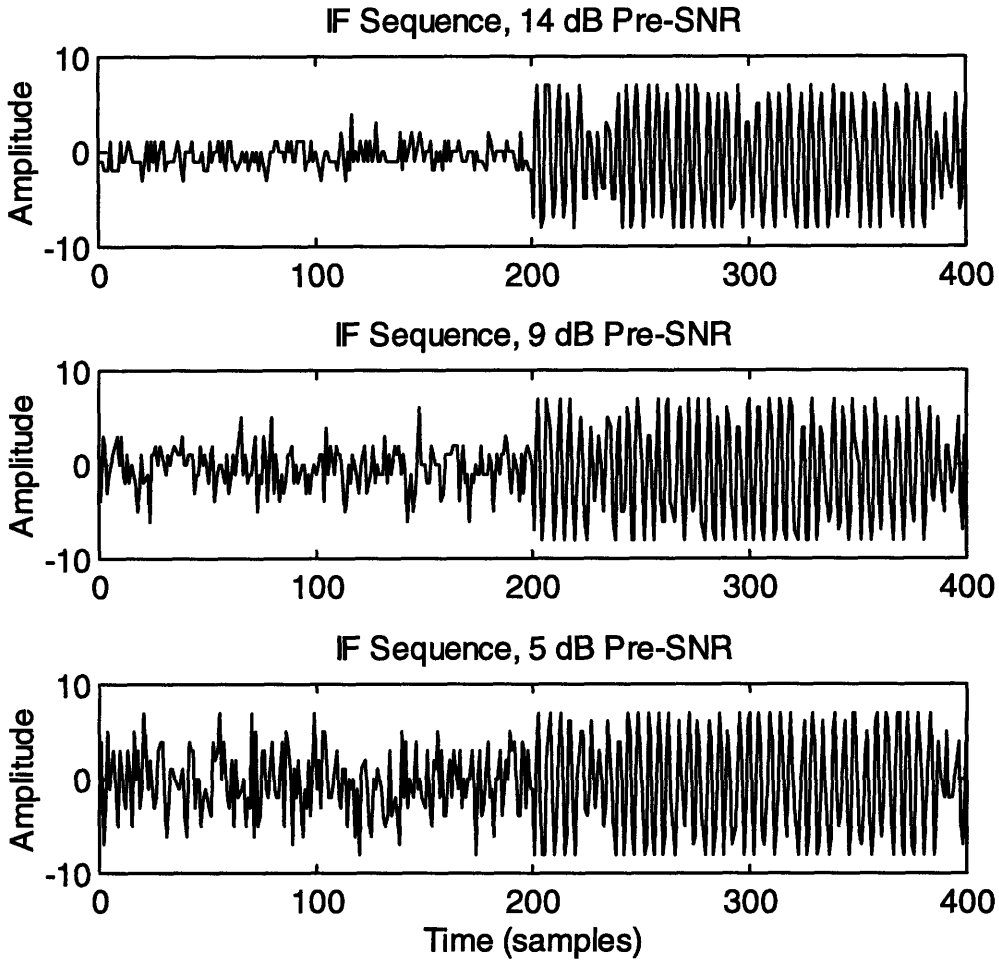


Figure 12-2 Pre-Transmission Channel Impairment

Figure 12-3 displays the results of noise added to real and imaginary parts of the modulated complex phasor. The figure displays the value of the complex phasor at the correct sampling offset in the raised cosine pulse shape. Thus, the plots resemble the odd and even subsets of the absolute phase constellation. The odd and even subsets are

displayed separately to emphasize that there are always effectively four decision regions. In the case of 9 dB transmit SNR, the full absolute phase constellation would seem very diffuse, even though the odd and even subsets are still recognizable. Note that the symbol phasors are more diffusely spread about the constellation points as noise power increases relative to signal power.

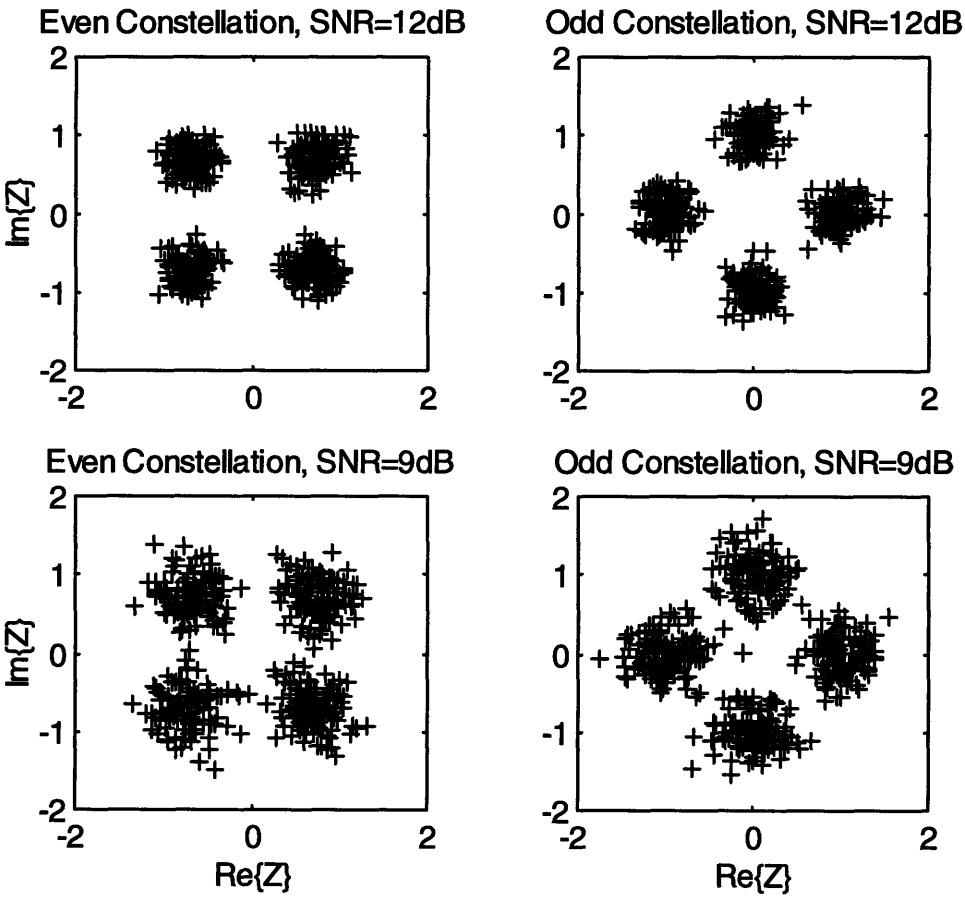


Figure 12-3 Additive Phasor Noise, Channel Impairment

12.2 Test Results

As mentioned earlier, simulation was performed to determine the expected amount of synchronization delay for the different designs. For each trial, several parameters were

recorded for the receive window that led ultimately to synchronization. Since there are 10 values of QI upon which the symbol timing decision is made, two important parameters are the maximum value of QI that determined the symbol timing choice, and the standard deviation of the QI set normalized to a maximum value of unity. The magnitude of the maximum QI is an indication of the amount of valid transmission present within the receive window. As the results table in the Appendix indicates, this value generally decreases as the window offset increases, rising again when the partially overlapping window fails, and the next window leads to synchronization. The standard deviation of normalized QI is of interest, since as noise increases and the amount of valid transmission within the window decreases, QI values become closer to one another, making the maximization decision less reliable. Thus the smaller standard deviations correspond to receive windows overlapping with little valid transmission. The standard deviations presented may be compared to the standard deviation obtained from the normalized QI presented in Figure 7-4. Since the symbol timing decision is actually performed over only 10 candidates, one of two possible decision sets is available out of 20 samples per symbol. The set among the two that actually matches the implementation, depends on the offset of the 20 samples-per-symbol to 10 samples-per-symbol down sampling procedure. However, the standard deviation in either case is approximately 0.41, compared to values ranging between 0.28 and 0.41 for the simulations. This is only a rough indicator of the metric since an ideal QI set with value one at the correct sampling position and zero elsewhere has a standard deviation of only 0.3. However, since pulse shaping has been used, the actual distribution of the QI set closely matches Figure 7-4. As can be seen in the result table, the values of maximum QI and standard deviation of normalized QI vary

greatly, since the particular noise sequences for each trial are generated independently. Trend behavior can only be apparent upon averaging the results from many trials at each offset.

Also, the frequency offset estimate yielded by symbol timing of the window leading to synchronization, is presented for each trial. The offset is recorded in kHz, scaled from the value represented in simulation to the appropriate units. It is apparent that the implementation has some bias to an expected value slightly lower than the actual value. However, it is also apparent that this bias is not significant, since a wide range of frequency offset estimates, as far off as 7.7 kHz on the low side, and 14.4 kHz on the high side, can still be driven to better estimates by carrier recovery loop convergence. Early Bellcore designs employed only a modified first order loop, in which the second order phase correction was initialized by the frequency offset, but not updated.[11] These results show that the more sophisticated second order loop extends the receiver performance into the regime where initial frequency offset estimates are poor.

Another parameter recorded is a Carrier Recovery Quality Indicator, (CRQI). This is just a sum of the backward loop error squared, symbol by symbol, for 15 symbols or half a window. Only the first 15 symbols of the backward loop error are used since they correspond temporally to the last half of the window. Thus, receive window offsets of less than 15 symbols, which would have high error energy in the end of the backward loop, do not impact CRQI. What results is a rough indicator of error energy, with higher CRQI at lower values of SNR. However, it is only a rough indicator as it is highly dependent on the particular noise sequence. Thus, as in the case of QI, it should only be

considered after averaging over many trials with independent noise sequences for a given offset.

Finally, the synchronization delay, or the time of synchronization in symbols relative to the start of the transmission, expresses the worst case overhead required. Also recorded is the synchronization word which actually lead to a correlation match.

The single half burst demodulator and the parallel half burst demodulator can now be compared along these parameters. Figure 12-4 summarizes the synchronization delay results for the various trials and leads to some interesting conclusions. The pattern of synchronization delays can be considered as points on several parallel delay curves. Each curve corresponds to synchronization occurring from the results of later windows. The advantages of the parallel half burst receiver over the single half burst receiver, and in turn over the full burst receiver, are now readily visible. The vertical spacing between delay curves corresponds to the time offset between successive windows. For a full burst receiver, there would be a 60 symbol separation, for the half burst receiver displayed there is a 30 symbol separation, and for the parallel case there is effectively a 10 symbol separation. For the simulations performed, each trial permitted synchronization by the beginning of the second transmitted burst. However, these plots provide insight into the condition where synchronization will not be possible by the second burst, and the condition where synchronization may be possible much earlier. Note that as the offset increases to the point where the initial window fails, the delay jumps discontinuously to a higher adjacent delay curve. The window offset at which the next delay curve reaches a sync delay of 60 symbols is especially significant. This offset reflects the maximum offset

at which a partially overlapping window must succeed in correlation to allow the desired sync delay. In the case of the single receiver, this is an offset of approximately 10 symbols. Whether the half burst demodulator can reliably synchronize given offsets of up to 10 symbols, must be verified with further simulation. For the parallel case, there is *very* high probability of synchronizing by the beginning of the second burst, if enough sync words are provided. Even in the zero offset case, if both the first and second paths fail in yielding correlation matches, both of which fully overlap the transmission, the third path can still yield a correlation match before the beginning of the second burst. However, this condition is extreme, and if shorter sync delay is anticipated, fewer sync words are required. Given the same critical window offset of greater than 10 symbols leading to failure, the parallel case has a worst case sync delay of just over 40 symbols. Furthermore, only 2 sync words are required to achieve this. This result is consistent with the limited simulation, which reveals that a single receiver requires 3 sync words with worst case delay approaching 1 burst, and that a parallel receiver requires only 2 sync words with worst case delay of only about 2/3 burst. Thus the parallel receiver trades increased implementation complexity for a lower probability of miss *or* lower overhead with limited backward buffering.

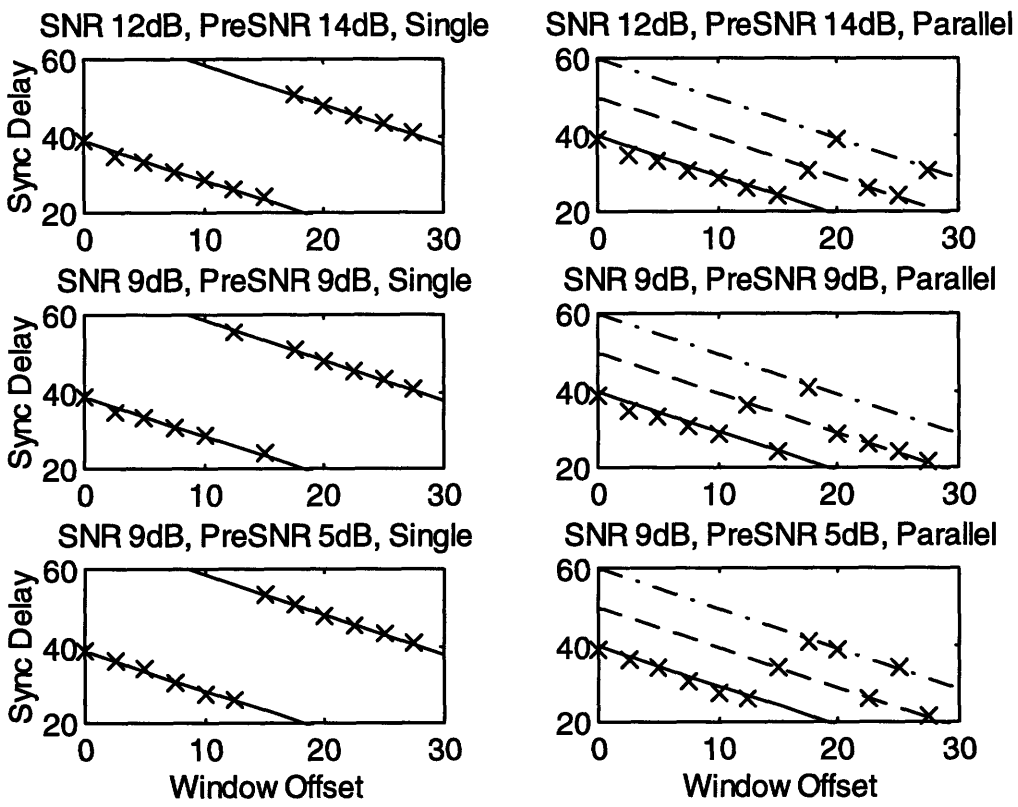


Figure 12-4 Synchronization Delay of different receiver designs.

13. Conclusion

13.1 Future Verification and Quantitative Estimates

Important insight into the required synchronization overhead is now available. However, more extensive simulation must be performed to determine metrics such as Prob.(miss) over a wide range of channel impairments. Furthermore, simulation is required to determine the important parameter of critical window offset, in order to quantitatively determine worst case sync delay. An estimate of Prob.(miss) as a function of window offset will allow one to obtain the critical window offset corresponding to the

maximum acceptable Prob.(miss). Given this critical offset, the plots of sync delay as a function of window offset can be used to determine worst case sync delay. The critical offset is the point at which sync delay moves from one delay curve to the next, and it is this discontinuous jump that leads to the worst case sync delay. Thus, knowledge of the critical offset allows one to determine whether the receiver enables sync by the second transmitted burst, for a given choice of Prob.(miss). It also allows one to determine the worst case delay for the parallel receiver, which will be less than one burst. The lower the delay for the parallel receiver, the more likely it becomes that only two sync words are required, and that less backward buffering is necessary.

Note that this design decision is driven by the external choice of acceptable Prob.(miss). This choice must be made with higher layer network performance considerations in mind.

13.2 Recommendations for Protocol Study

The fundamental decision of which receiver design is most appropriate depends heavily on higher layer protocol decisions. The influence of Prob.(miss) determination on worst case sync overhead has already been discussed. In addition, if a polling strategy is employed, the receiver demodulate and decode latency is no longer a driving concern. If long transmissions are employed, the modest decrease in sync overhead as a proportion of the transmission, may not justify the large increase in complexity of the parallel receiver. Thus, this work cannot alone determine the preferred receiver design, and more detailed protocol study is required to choose between the alternatives presented.

13.3 Summary

Packet radio applications such as mobile networks motivate the study of asynchronous receivers. Past work with burst coherent receivers in isochronous applications has shown their attractive error performance. However, in adapting Bellcore's burst coherent receiver for asynchronous use conformant with FCC rules, some formidable challenges are faced. A technique for reducing the receiver's processing latency has been shown, reducing transmission overhead from 6 bursts to 2. Further techniques, shortening the demodulation window to half a burst, may reduce transmission overhead further to only one burst. Moderate parallelism may be employed to reduce overhead to approximately 2/3 of a burst, or may be employed to allow lower Prob.(miss) sync with one burst of overhead. Finally, future simulation and protocol study must be performed to determine the receiver best suited for a given application.

14. References

[1] G. Pollini, "Media Access Control Requirements for personal communications services (PCS) devices operating in the 1910-1930 MHz unlicensed spectrum", Engineer's Notes, September 27, 1996.

[2] A. R. Noerpel, "A United States Perspective for a Flexible PCS Standard", ITU Telecom 95 Technology Summit, Geneva, October 3-11, 1995.

[3] J. A. C. Bingham, "Quadrature Modulation and Keying", *Theory and Practice of Modem Design*, pp. 81-83, 1988.

[4] N. Sollenberger, J. C-I Chuang, L. F. Chang, S. Ariyavisitakul, and H. W. Arnold, "Architecture and Implementation of an Efficient and Robust TDMA Frame Structure for Digital Portable Communications," *Proceedings, IEEE VTC'89II*, pp. 169-174, May 1989.

[5] J. C-I Chuang and N. Sollenberger, "Burst Coherent Detection With Robust Frequency and Timing Estimation For Portable Radio Communications," Proceedings, IEEE GLOBECOM'88, Hollywood, Florida, November 1988, pp. 804-809.

[6] G. Pollini, "Physical layer services provided to the Media Access Control (MAC) layer by the PACS prototype radio for operation in the 1910-1930 MHz unlicensed spectrum.", Engineer's Notes, November 27, 1996.

[7] J. C-I Chuang and N. Sollenberger, "Burst Coherent Detection With Robust Frequency and Timing Estimation For Portable Radio Communications," Proceedings, IEEE GLOBECOM'88, Hollywood, Florida, November 1988, pp. 804-809.

[8] J. C-I Chuang and N. Sollenberger, "A High Performance Diversity Selection Technique for TDMA Portable Radio Communications," Record, IEEE GLOBECOM'89, Dallas, Texas, Nov. 27-30, 1989 pp.1361-1365.

[9] N. R. Sollenberger, and J. C-I Chuang "Low Overhead Symbol Timing and Carrier Recovery for TDMA Portable Radio Systems," Proceedings, Third Nordic Seminar on Digital Land Mobile Radio Communication, Copenhagen, Denmark, September 1988, Paper #10.3.

[10] J. C-I Chuang and N. Sollenberger, "Burst Coherent Detection With Robust Frequency and Timing Estimation For Portable Radio Communications," Proceedings, IEEE GLOBECOM'88, Hollywood, Florida, November 1988, pp. 804-809.

[11] N. R. Sollenberger, and J. C-I Chuang "Low Overhead Symbol Timing and Carrier Recovery for TDMA Portable Radio Systems," Proceedings, Third Nordic Seminar on Digital Land Mobile Radio Communication, Copenhagen, Denmark, September 1988, Paper #10.3.

Appendix A

Tables of Results

HALF BURST DEMOD

Metrics for window leading to sync.

Xmit SNR	Pre-Xmit SNR	Frequency Offset	Receive Window Offset	Std(QI)	Max(QI)	Std(QI)/Max(QI)	Estimated Frequency Offset	CRQI
12	14	12	0.0	11.47	30	0.3823	11.3	935
12	14	12	2.5	10.67	28	0.3811	11.3	691
12	14	12	5.0	9.20	25	0.3680	12.0	796
12	14	12	7.5	6.85	24	0.2854	10.3	695
12	14	12	10.0	5.45	16	0.3406	10.9	1275
12	14	12	12.5	3.06	11	0.2782	14.4	867
12	14	12	15.0	4.48	16	0.2800	10.9	611
12	14	12	17.5	14.52	43	0.3377	10.7	610
12	14	12	20.0	11.91	31	0.3842	10.5	966
12	14	12	22.5	11.64	31	0.3755	10.5	846
12	14	12	25.0	13.63	36	0.3786	11.3	670
12	14	12	27.5	11.93	31	0.3848	10.5	799
9	9	12	0.0	11.73	30	0.3910	11.3	863
9	9	12	2.5	6.47	19	0.3405	10.3	1491
9	9	12	5.0	5.16	14	0.3686	10.9	2779
9	9	12	7.5	5.17	14	0.3693	12.0	1781
9	9	12	10.0	5.02	18	0.2789	9.2	1830
9	9	12	12.5	6.28	20	0.3140	11.1	1179
9	9	12	15.0	5.51	19	0.2900	10.3	815
9	9	12	17.5	4.98	16	0.3113	10.9	1270
9	9	12	20.0	8.06	21	0.3838	10.3	1555
9	9	12	22.5	8.35	25	0.3340	11.1	1280
9	9	12	25.0	11.19	29	0.3859	10.5	998
9	9	12	27.5	11.22	33	0.3400	11.3	1053
9	5	12	0.0	10.32	34	0.3035	10.7	823
9	5	12	2.5	7.56	26	0.2908	10.3	868
9	5	12	5.0	3.78	11	0.3436	10.9	1931
9	5	12	7.5	4.84	16	0.3025	12.9	892
9	5	12	10.0	2.90	10	0.2900	9.4	1873
9	5	12	12.5	3.73	11	0.3391	12.0	895
9	5	12	15.0	9.43	29	0.3252	10.5	1413
9	5	12	17.5	9.70	24	0.4042	11.0	2199
9	5	12	20.0	7.47	23	0.3248	11.1	1505
9	5	12	22.5	7.17	20	0.3585	12.0	1578
9	5	12	25.0	7.80	21	0.3714	10.3	1038
9	5	12	27.5	10.70	26	0.4115	10.3	1215

HALF BURST DEMOD

Xmit SNR	Pre-Xmit SNR	Frequency Offset	Receive Window Offset	Sync Time (Symbols)	Sync Word	Post Sync Decode bit errors, (out of 15, 90-bit payloads
12	14	12	0.0	38.8	2	0
12	14	12	2.5	34.9	1	0
12	14	12	5.0	33.1	1	0
12	14	12	7.5	30.4	1	0
12	14	12	10.0	28.6	1	0
12	14	12	12.5	25.9	1	0
12	14	12	15.0	24.1	1	0
12	14	12	17.5	50.8	3	0
12	14	12	20.0	47.8	2	0
12	14	12	22.5	45.1	2	0
12	14	12	25.0	43.3	2	0
12	14	12	27.5	40.6	2	0
9	9	12	0.0	38.8	2	0
9	9	12	2.5	34.9	1	0
9	9	12	5.0	33.1	1	0
9	9	12	7.5	30.4	1	0
9	9	12	10.0	28.6	1	3
9	9	12	12.5	55.3	3	0
9	9	12	15.0	24.1	1	0
9	9	12	17.5	50.8	3	0
9	9	12	20.0	47.8	2	0
9	9	12	22.5	45.1	2	1
9	9	12	25.0	43.3	2	0
9	9	12	27.5	40.6	2	0
9	5	12	0.0	38.8	2	0
9	5	12	2.5	36.1	2	0
9	5	12	5.0	34.3	2	0
9	5	12	7.5	30.4	1	0
9	5	12	10.0	27.7	1	0
9	5	12	12.5	25.9	1	0
9	5	12	15.0	53.5	3	0
9	5	12	17.5	50.8	3	0
9	5	12	20.0	47.8	2	0
9	5	12	22.5	45.1	2	0
9	5	12	25.0	43.3	2	2
9	5	12	27.5	40.6	2	0

PARALLEL HALF BURST DEMOD

Metrics for window leading to sync.

Xmit SNR	Pre-Xmit SNR	Frequency Offset	Receive Window Offset	Std(QI)	Max(QI)	Std(QI)/Max(QI)	Estimated Frequency Offset	CRQI
12	14	12	0.0	11.47	30	0.3823	11.3	935
12	14	12	2.5	10.67	28	0.3811	11.3	691
12	14	12	5.0	9.20	25	0.3680	12.0	796
12	14	12	7.5	6.85	24	0.2854	10.3	695
12	14	12	10.0	5.45	16	0.3406	10.9	1275
12	14	12	12.5	3.06	11	0.2782	14.4	867
12	14	12	15.0	4.48	16	0.2800	10.9	611
12	14	12	17.5	5.87	20	0.2935	11.1	1260
12	14	12	20.0	14.97	40	0.3743	10.7	808
12	14	12	22.5	3.84	12	0.3200	12.0	843
12	14	12	25.0	3.07	8	0.3838	7.7	882
12	14	12	27.5	5.02	15	0.3347	10.1	834
9	9	12	0.0	11.73	30	0.3910	11.3	863
9	9	12	2.5	6.47	19	0.3405	10.3	1491
9	9	12	5.0	5.16	14	0.3686	10.9	2779
9	9	12	7.5	5.17	14	0.3693	12.0	1781
9	9	12	10.0	5.02	18	0.2789	9.2	1830
9	9	12	12.5	8.26	21	0.3933	10.3	1213
9	9	12	15.0	5.51	19	0.2900	10.3	815
9	9	12	17.5	8.03	23	0.3491	12.0	1901
9	9	12	20.0	6.24	15	0.4160	10.1	1053
9	9	12	22.5	2.69	8	0.3363	7.7	1453
9	9	12	25.0	2.51	9	0.2789	9.4	1054
9	9	12	27.5	2.32	8	0.2900	13.1	1281
9	5	12	0.0	10.32	34	0.3035	10.7	823
9	5	12	2.5	7.56	26	0.2908	10.3	868
9	5	12	5.0	3.78	11	0.3436	10.9	1931
9	5	12	7.5	4.84	16	0.3025	12.9	892
9	5	12	10.0	2.90	10	0.2900	9.4	1873
9	5	12	12.5	3.73	11	0.3391	12.0	895
9	5	12	15.0	6.29	16	0.3931	10.9	1632
9	5	12	17.5	11.02	31	0.3555	10.5	1094
9	5	12	20.0	5.94	19	0.3126	12.8	2907
9	5	12	22.5	3.86	11	0.3509	8.4	1093
9	5	12	25.0	2.97	8	0.3713	13.1	1325
9	5	12	27.5	4.86	16	0.3038	12.9	1252

PARALLEL HALF BURST DEMOD

Xmit SNR	Pre-Xmit SNR	Frequency Offset	Receive Window Offset	Sync Time (Symbols)	Sync Word	Post Sync Decode errors, (out of 15, 90-bit payloads
12	14	12	0.0	38.8	2	0
12	14	12	2.5	34.9	1	0
12	14	12	5.0	33.1	1	0
12	14	12	7.5	30.4	1	0
12	14	12	10.0	28.6	1	0
12	14	12	12.5	25.9	1	0
12	14	12	15.0	24.1	1	0
12	14	12	17.5	30.4	1	0
12	14	12	20.0	38.8	2	0
12	14	12	22.5	25.9	1	0
12	14	12	25.0	24.1	1	0
12	14	12	27.5	30.4	1	0
9	9	12	0.0	38.8	2	0
9	9	12	2.5	34.9	1	0
9	9	12	5.0	33.1	1	0
9	9	12	7.5	30.4	1	0
9	9	12	10.0	28.6	1	3
9	9	12	12.5	36.1	2	0
9	9	12	15.0	24.1	1	0
9	9	12	17.5	40.6	2	0
9	9	12	20.0	28.6	1	0
9	9	12	22.5	25.9	1	1
9	9	12	25.0	24.1	1	0
9	9	12	27.5	21.4	1	0
9	5	12	0.0	38.8	2	0
9	5	12	2.5	36.1	2	0
9	5	12	5.0	34.3	2	0
9	5	12	7.5	30.4	1	0
9	5	12	10.0	27.7	1	0
9	5	12	12.5	25.9	1	0
9	5	12	15.0	34.3	2	0
9	5	12	17.5	40.6	2	0
9	5	12	20.0	38.8	2	0
9	5	12	22.5	25.9	2	0
9	5	12	25.0	34.3	2	2
9	5	12	27.5	21.4	1	0

Appendix B

Simulation Code

Generating IF Stream Input

`gadn.m`	script to generate IF stream, input: source bit stream, output: IF stream
`rc.m`	function to generate raised cosine time domain pulse shape
`makecw.m`	function to generate PACS 105,90 codeword given 90 bit payload

Candidate Receiver Functions

`runrcv.m`	script to run large latency full burst receiver
`runhlf.m`	script to run half burst receiver and fast full burst receiver after sync
`runpll.m`	script to run parallel half burst rcvr and fast full burst receiver after sync
`receive.m`	function for large latency full burst receiver
`hlfrcv.m`	function for half burst receiver
`pllhlf.m`	function for parallel staggered half burst receiver
`fastrcv.m`	function for fast full burst receiver
`symtimfo.m`	function for full burst symbol timing, carrier offset estimation
`carrec.m`	function for full burst carrier recovery
`symtmhlf.m`	function for half burst symbol timing, carrier offset estimation
`hlfcr.m`	function for half burst carrier recovery
`correl.m`	function for correlator
`fdecode.m`	function for set of staggered CRC decoders
`errchk.m`	function for individual CRC decoder

Generating Function Look Up Tables

`atan10x6.m`	script to generate arctan lookup used in frequency offset estimation
`atan8x6.m`	script to generate arctan lookup used in band pass to phase conversion
`cos4t6x8.m`	script to generate cosine lookup used in QI computation
`sin4t6x8.m`	script to generate sine lookup used in QI computation
`sqr6x8.m`	script to generate square lookup used in QI computation

Utility Functions

`di2bits.m`	function to convert dibit stream {0,1,2,3} to bit stream twice as long {0,1}
`do2scomp.m`	function to convert 2s complement representation to signed representation
`modulo.m`	function for modulo division

GADN.M

```
%gadn.m
%
% generates input stream for demodulator
% 4 bits/sample, 20 samples/symbol
% raised cosine pulse shaping
% burst is now centered between guard times!
% additive white gaussian noise included

% INPUTS: Vectors of (1 element per burst) :
%       -SNR
%
%       Vectors of bits
%       -payload, series of 90 bit payloads

% raised cosine pulse shape
p=rc((-60:59)/20,0.5);

% phase mapper
% dibit         phase change
% -----         ------------
%  00             pi/4
%  01            3pi/4
%  10            -pi/4
%  11           -3pi/4

dphmap=[1,3,-1,-3];

%clip=input('Clipping fraction of peak-to-peak? ');
clip=1;

paramstr=input('Enter file name (w/o extension) for Matlab parameter file: ','s');
% this file has to have the variables:
% foff, snrvec, bitstrm, offset (between 1 and 1200), dedsnr

eval(['load ', paramstr]);

bl=length(bitstrm);
nburst=bl/120;
if (modulo(bl,120)>0)
        disp('Bit stream length is not integer multiple of 120 bits.');
        error('Wrong length!');
elseif ~(nburst==length(snrvec))
        disp('Payload length does not match SNR vector.');
        error('Wrong length!');
end

sessionstr=input('Enter file name (w/o extension) for Matlab session file: ','s');

% pre-size some variables
currburst=zeros(1,120);
dphseq=zeros(1,nburst*60);
phseq=zeros(1,nburst*60);
```

```
bbseq=zeros(1,nburst*60*20);
tempc=zeros(1,nburst*60*20+120-1);
fbbseq=zeros(1,nburst*60*20);
fbbseqn=zeros(1,nburst*60*20);
ifseq=zeros(1,nburst*60*20);
deadair=zeros(1,offset);
ifseqd=zeros(1,offset+nburst*60*20);
bitvec=zeros(1,960);

for burstcnt=1:nburst,

        % generate the sequence

        % read bit stream 120 bits at time
        currburst=bitstrm(((burstcnt-1)*120 + 1) : ((burstcnt-1)*120 + 120));

        % add crc to payload
        currburst(15:119)=makecw(currburst(15:104));

        % mark first and last codeword bits, pad first 2 bits w/ de symbol, add pc bit

        currburst(120)=1;          % pad bit set to 1

        % mark cw
        currburst(15)=~currburst(15);
        currburst(119)=~currburst(119);

        % now break it up into dibits! (single column per symbol)
        dphseq((burstcnt-1)*60+1:(burstcnt-1)*60+60) = dphmap(:, ones(1,60) + 2*currburst(1:2:120) +
currburst(2:2:120));
end


% calculate phase sequence; arbitrary 0 init
% use cumulative sum concept to get sequence
phseq=pi/4*modulo(cumsum(dphseq),8);

% now go to baseband (complex rep), dirac delta model one samp per baud
% ignoring the issue of sample phase alignment at this point . . .
bbseq(1:20:length(bbseq))=exp(j*phseq);

% interpolate to T/20 sampling and do filtering
tempc=conv(bbseq,p);

% strip leading and trailing bits from convolution
% length=nburst*1200+120-1 so subtract 119, 60 before and 59 after
fbbseq=tempc(61:length(tempc)-59);


% ADD AWGN PROCESS . . . do BEFORE upconvert to mimic bandpass noise!!!
% calibrate "power" of fbbseq
sqpow=sqrt((fbbseq*fbbseq')/nburst/1200);
sigmavec=10.^(-snrvec/20)*sqpow;
```

```
% remember complex so need to divide sigma by sqrt(2) for I & Q to get right SNR
% complex noise, add to I and Q
randn('seed',sum(100*clock));
inoise=randn(1,1200*nburst);
randn('seed',sum(100*clock));
qnoise=randn(1,1200*nburst);
for mmm=1:nburst,
        inoise((mmm-1)*1200 + 1 : (mmm-1)*1200 + 1200) = sigmavec(mmm)/sqrt(2)*inoise((mmm-
1)*1200 + 1 : (mmm-1)*1200 + 1200);
        qnoise((mmm-1)*1200 + 1 : (mmm-1)*1200 + 1200) = sigmavec(mmm)/sqrt(2)*qnoise((mmm-
1)*1200 + 1 : (mmm-1)*1200 + 1200);
end
fbbseqn=fbbseq+inoise+j*qnoise;

% now mix up to IF
% the actual IF frequency is -820 kHz (10700 - 3*3840)
% represents as not 1/4 but -41/192
ifseq=real(exp(j*2*pi*(-41/192 + foff/3840)*(0:nburst*1200-1)) .* fbbseqn);

% now preface w/ dead air w/ gwn
dedpow=sqrt((ifseq(1:1200)*ifseq(1:1200)')/1200);
dedsigma=10.^(-dedsnr/20)*dedpow;
deadair=dedsigma*randn(1,offset);


% do 4 bit quantization
ifseqd=floor(8/clip*[deadair, ifseq]);
cliphi=find(ifseqd>7); shi=length(cliphi); ifseqd(cliphi)=7*ones(1,shi);
cliplo=find(ifseqd<-8); slo=length(cliplo); ifseqd(cliplo)=-8*ones(1,slo);

% NOTE: demod input is in signed format

% Save matlab variables
eval(['save ',sessionstr])
```

RC.M

```
function yyy=rc(xxx,alpha)
%
% calculates RC pulse shape in the time domain
% given values of t/T, alpha
% note NOT scaled by 1/T
b=size(xxx);
fact=cos(pi*alpha*xxx)./(ones(b(1),b(2))-(2*alpha*xxx).^2);
% check for special cases
spec=find(abs(xxx) == 0.5/alpha);
c=size(spec);
fact(spec)=ones(c(1),c(2))*pi/4;
yyy=sinc(xxx).*fact;
```

MAKECW.M

```
function cw=makecw(bitvec);
%
% calculates CRC for PACS channel code given
% payload (slow and fast channel bits)
%
% payload given as 90 bit vector; [bit0 bit1 ... bit89]
% CRC is appended and 105 bit code word is returned

% do long division
% initialize, add 15 zeros to effect multiply by x^15
dividend=[bitvec(1:90),zeros(1,15)];

divisor=[1,1,1,1,1,0,1,1,0,1,0,0,0,0,0,1]; % PACS standard generator

done=0;

while ~done,
        % determine the position for the quotient
        % pos=k -> dividend has term at x^(105-k)
        pos=min(find(dividend==1));
        if pos <= 90,
                % subtract the appropriate multiple of the divisor
                sterm=[zeros(1,pos-1),divisor,zeros(1,90-pos)];
                dividend=modulo(dividend+sterm,2);
        else
                % we're done!
                done=1;
        end     % end if
end     % end while

% now what is left of the dividend is the CRC
% (the last 15 bits)
crc=[dividend(91:105)];
cw=[bitvec(1:90),crc(1:15)];
return;
```

RUNRCV.M

```
[tptrall,weiall,slipall,payall,syncall,detectall,bitsall,fphaseall,ffall,ferrall,bphaseall,bfall,berrall,CRQIall,dp
h8all,gateph8all,nmetricall,nfreqall,maxall,ph8lvec]=receive(ifseqd,atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x
8);
```

RUNHLF.M

```
[detect1all,detect2all,detect3all,detect4all,detect5all,bits1all,fphase1all,ff1all,ferr1all,bphase1all,bf1all,ber
r1all,CRQI1all,dph81all,gateph81all,nmetric1all,nfreq1all,max1all,ph8l1vec,newptr]=hlfrcv(ifseqd,atn8x
6,atn10x6,sq6x8,cs4t6x8,sn4t6x8);
[weiall,slipall,payall,bitsall,fphaseall,ffall,ferrall,bphaseall,bfall,berrall,CRQIall,dph8all,gateph8all,nmetri
call,nfreqall,maxall,ph8lvec]=fastrcv(ifseqd,newptr,atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8);
```

RUNPLL.M

```
[detect11all,detect21all,nmetric1all,nfreq1all,max1all,CRQI1all,detect12all,detect22all,nmetric2all,nfreq2
all,max2all,CRQI2all,detect13all,detect23all,nmetric3all,nfreq3all,max3all,CRQI3all,newptr]=pllhlf(ifseq
d,atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8);
[weiall,slipall,payall,bitsall,fphaseall,ffall,ferrall,bphaseall,bfall,berrall,CRQIall,dph8all,gateph8all,nmetri
call,nfreqall,maxall,ph8lvec]=fastrcv(ifseqd,newptr,atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8);
```

RECEIVE.M

```
function
[tptrall,weiall,slipall,payall,syncall,detectall,bitsall,fphaseall,ffall,ferrall,bphaseall,bfall,berrall,CRQIall,dp
h8all,gateph8all,nmetricall,nfreqall,maxall,ph8lvec]=receive(ifseqd,atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x
8)
%
% Parallel implementation of demodulator code.
% Symbol timing implemented one burst vector at a time.
% Coherent recovery still uses serial looping.
% 2nd order carrier recovery . . . "fast" poles
% presumes 10.7 MHz input IF, sampled at 3840 kHz, associated compensations



% CORRELATION PATTERN

%corrpat=[0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1];
corrpat=[0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1];  % must don't care symbol 0 (diff'l encoding)


% SYM TIM CONSTANT VECTORS

invert=1;                % bit to account for any inversions in IF

% IF compensation to add 140 kHz
% done as 9,9,10,9,9,10 brad per sample! (9,9,9,... would give only 135 kHz)
ifcomp=[10;9;9]*ones(1,400); ifcomp=ifcomp(:);

% deal with wraparound
ifcomp=do2scomp(cumsum(ifcomp)',8);

% filter coefficients (what about Rob's extra latch in bpphase2.m?)
bpfilt=[1,0,-1];
xfilt=[0,1,0,-2,0,1];
yfilt=[0,0,2,0,-2,0];

% make mod 4 counter (vector of 1200)
cnt=modulo(1:1200,4);
cnt1=floor(cnt/2);
cnt0=modulo(cnt,2);

start=7;                 % Start symbol of accumulation window in burst
finish=53;               % Stop symbol of accumulation window in burst


% CARRIER RECOVERY CONSTANTS

alpha=1/2;               % alpha gain in 2nd order loop
beta=1/16;               % beta gain in 2nd order loop



% if sequence is signed format (not 2s comp)
```

```
tptr=1;
rptr=1;
sync=0;
index=0;

% initialize delay variables

% Symbol Timing
stmaxd=zeros(1,3);
dph8vecd=zeros(1,60);
nmetricd=zeros(10,1);
nfreqd=zeros(10,1);
stPLLd=zeros(1,4);
gateph8d=zeros(1,60);

% Carrier Recovery
streordd=zeros([1 30]);    % reorder RAM, inputs: {b,f}phaser{2}msb, # states: 30
dout=zeros(1,60);

% Sync Slip Error Detect
payloadd=zeros(1,90);
weid=0;
slipd=0;



% check input sequence
rl=length(ifseqd);


while tptr <= (floor(rl/1200)-1)*1200+1,

        index=index+1;

        % take a burst of data from input stream
        curr_burst=ifseqd(tptr:tptr+1199);

        [stmax,stPLL,dph8vec,gateph8,nmetric,nfreq,ph8l]=symtimfo(bpfilt,xfilt,yfilt,cnt1,cnt0,atn8x6,at
n10x6,sq6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,curr_burst);

        [ffvec,fphasevec,bfvec,bphasevec,ferrvec,berrvec,stCRQI,streord,dout]=carrec(alpha,beta,streordd
,stPLLd,gateph8d);

        corrdat=di2bits(dout);

        detect=correl(corrdat,corrpat);

        [wei,slip,payload]=fdecode(corrdat);

        %TEMP
        ph8lvec((index-1)*1200+1:(index-1)*1200+1200)=ph8l;

        %% Compose diagnostic output vectors
```

```
tptrall(index)=tptr;
weiall(index)=weid;
slipall(index)=slipd;
payall((index-1)*90+1:(index-1)*90+90)=payloadd;
syncall(index)=sync;
detectall(index)=detect;
bitsall((index-1)*120+1:(index-1)*120+120)=corrdat;
fphaseall((index-1)*60+1:(index-1)*60+60)=fphasevec;
ffall((index-1)*60+1:(index-1)*60+60)=ffvec;
ferrall((index-1)*60+1:(index-1)*60+60)=ferrvec;
bphaseall((index-1)*60+1:(index-1)*60+60)=bphasevec;
bfall((index-1)*60+1:(index-1)*60+60)=bfvec;
berrall((index-1)*60+1:(index-1)*60+60)=berrvec;
CRQIall(index)=stCRQI;
dph8all((index-1)*60+1:(index-1)*60+60)=dph8vecd;
gateph8all((index-1)*60+1:(index-1)*60+60)=gateph8d;
nmetricall(:,index)=nmetricd;
nfreqall(:,index)=nfreqd;
maxall(index,:)=stmaxd;


%%%%% Delay elements to represent pipeline latencies follow
% Symbol Timing
stmaxd=stmax;
dph8vecd=dph8vec;
nmetricd=nmetric;
nfreqd=nfreq';
stPLLd=stPLL;
gateph8d=gateph8;

% Carrier Recovery
% NOTE: forward and backward frequency, phase and error vectors
% are not delayed, since they are "instantaneous" outputs
streordd=streord;  % ends up delaying dout

% Sync Slip Error Detect
payloadd=payload;
weid=wei;
slipd=slip;


if sync==0,
        if detect
                % remember pattern is 1 symbol short
                rptr=(detect-(length(corrpat)+2)+6)*10+tptr+1200;
                if rptr==tptr+1200,
                        sync=2;
                else
                        sync=1;
                end
        end
        tptr=tptr+1200;
elseif sync==1,
        % truncate overprocessed streams, clear pipes (actually pipes flushed by next stream)
```

```
                payall=payall(1:(index-1)*90); % swallow fractional window output
                bitsall=bitsall(1:(index-1)*120+detect-(length(corrpat)+2)+6);
                fphaseall=fphaseall(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));
                ffall=ffall(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));
                ferrall=ferrall(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));
                bphaseall=bphaseall(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));
                bfall=bfall(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));
                berrall=berrall(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));
                dph8all=dph8all(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));
                gateph8all=gateph8all(1:(index-1)*60+ceil((detect-(length(corrpat)+2)+6)/2));

                % Symbol Timing
                stmaxd=zeros(1,3);
                dph8vecd=zeros(1,60);
                nmetricd=zeros(10,1);
                nfreqd=zeros(10,1);
                stPLLd=zeros(1,4);
                gateph8d=zeros(1,60);

                % Carrier
                streordd=zeros([1 30]);
                dout=zeros(1,60);

                % Sync Slip Error Detect
                payloadd=zeros(1,90);
                weid=0;
                slipd=0;

                tptr=rptr;
                sync=2;
        elseif sync==2,
                tptr=tptr+1200;
        end

end

return;
```

HLFRCV.M

```
function
[detect1all,detect2all,detect3all,detect4all,detect5all,bitsall,fphaseall,ffall,ferrall,bphaseall,bfall,berrall,CR
QIall,dph8all,gateph8all,nmetricall,nfreqall,maxall,ph8lvec,newptr]=hlfrcv(ifseqd,atn8x6,atn10x6,sq6x8,
cs4t6x8,sn4t6x8)
%
% Parallel implementation of demodulator code.
% Symbol timing implemented one burst vector at a time.
% Coherent recovery still uses serial looping.
% 2nd order carrier recovery . . . "fast" poles
% presumes 10.7 MHz input IF, sampled at 3840 kHz, associated compensations



% ONLY PURPOSE IS TO ACQUIRE SYNC, AFTER A DETECT RESET AND CALL OUT TO FULL
BURST DEMOD

% CORRELATION PATTERNS

corrpat1=[0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 1 0 1];  % must don't care symbol 0 (diff'l encoding)
corrpat2=[0 0 1 1 1 0 1 1 0 0 0 0 1 0 0 0 0 0 1 1 0 1];
corrpat3=[0 0 1 1 1 0 1 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1];
corrpat4=[0 0 1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 1];
corrpat5=[0 0 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 0 1 1 0 1];


% SYM TIM CONSTANT VECTORS

invert=1;                    % bit to account for any inversions in IF

% IF compensation to add 140 kHz
% done as 9,9,10,9,9,10 brad per sample! (9,9,9,... would give only 135 kHz)
ifcomp=[10;9;9]*ones(1,200); ifcomp=ifcomp(:);

% deal with wraparound
ifcomp=do2scomp(cumsum(ifcomp)',8);

% filter coefficients (what about Rob's extra latch in bpphase2.m?)
bpfilt=[1,0,-1];
xfilt=[0,1,0,-2,0,1];
yfilt=[0,0,2,0,-2,0];

% make mod 4 counter (vector of 1200)
cnt=modulo(1:600,4);
cnt1=floor(cnt/2);
cnt0=modulo(cnt,2);

start=0;                     % Start symbol of accumulation window in burst
finish=30;                   % Stop symbol of accumulation window in burst


% CARRIER RECOVERY CONSTANTS
```

```
alpha=1/2;              % alpha gain in 2nd order loop
beta=1/16;              % beta gain in 2nd order loop


% if sequence is signed format (not 2s comp)

tptr=1;
index=0;
rate=10;  % rate multiplication for fast carrier recovery


% initialize delay variables

% Symbol Timing
stmaxd=zeros(1,3);
dph8vecd=zeros(1,30);
nmetricd=zeros(10,1);
nfreqd=zeros(10,1);
stPLLd=zeros(1,4);
gateph8d=zeros(1,30);

% Carrier Recovery
dout=zeros(1,30);

% check input sequence
rl=length(ifseqd);

sync=0;

while ~(sync),

        index=index+1;

        % take a half burst of data from input stream
        curr_burst=ifseqd(tptr:tptr+599);

        [stmax,stPLL,dph8vec,gateph8,nmetric,nfreq,ph8l]=symtmhlf(bpfilt,xfilt,yfilt,cnt1,cnt0,atn8x6,a
tn10x6,sq6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,curr_burst);

        [ffvec,fphasevec,bfvec,bphasevec,ferrvec,berrvec,stCRQI,dout]=hlfcr(alpha,beta,stPLLd,gateph8d
);

        corrdat=di2bits(dout);

        % streaming correlator should work across window slices
        if index>1,
                detect1=correl([bitsall((index-2)*60+38:(index-2)*60+60),corrdat],corrpat1);
                detect2=correl([bitsall((index-2)*60+38:(index-2)*60+60),corrdat],corrpat2);
                detect3=correl([bitsall((index-2)*60+38:(index-2)*60+60),corrdat],corrpat3);
                detect4=correl([bitsall((index-2)*60+38:(index-2)*60+60),corrdat],corrpat4);
                detect5=correl([bitsall((index-2)*60+38:(index-2)*60+60),corrdat],corrpat5);
        else
                detect1=0;
                detect2=0;
```

```
        detect3=0;
        detect4=0;
        detect5=0;
end

%TEMP
ph8lvec((index-1)*600+1:(index-1)*600+600)=ph8l;

%% Compose diagnostic output vectors
tptrall(index)=tptr;
syncall(index)=sync;
detect1all(index)=detect1;
detect2all(index)=detect2;
detect3all(index)=detect3;
detect4all(index)=detect4;
detect5all(index)=detect5;
bitsall((index-1)*60+1:(index-1)*60+60)=corrdat;
fphaseall((index-1)*30+1:(index-1)*30+30)=fphasevec;
ffall((index-1)*30+1:(index-1)*30+30)=ffvec;
ferrall((index-1)*30+1:(index-1)*30+30)=ferrvec;
bphaseall((index-1)*30+1:(index-1)*30+30)=bphasevec;
bfall((index-1)*30+1:(index-1)*30+30)=bfvec;
berrall((index-1)*30+1:(index-1)*30+30)=berrvec;
CRQIall(index)=stCRQI;
dph8all((index-1)*30+1:(index-1)*30+30)=dph8vecd;
gateph8all((index-1)*30+1:(index-1)*30+30)=gateph8d;
nmetricall(:,index)=nmetricd;
nfreqall(:,index)=nfreqd;
maxall(index,:)=stmaxd;


%%%%% Delay elements to represent pipeline latencies follow
% Symbol Timing
stmaxd=stmax;
dph8vecd=dph8vec;
nmetricd=nmetric;
nfreqd=nfreq';
stPLLd=stPLL;
gateph8d=gateph8;

% Carrier Recovery
% NOTE: forward and backward frequency, phase and error vectors
% are not delayed, since they are "instantaneous" outputs
% NONE REQUIRED, since assuming it is run fast enough to complete
% in less that half burst, maybe even in 15 symbols...


[detect1, detect2, detect3, detect4, detect5]

if (detect1)
        start=tptr + (detect1-23-23)*10 - 600 % abs. reference (subt symtim latency)
        pos=tptr - start + 60/rate*20 + (detect1-23)/rate*10  % relative pos to start
elseif (detect2)
        start=tptr + (detect2-23-47)*10 - 600 % abs. reference (subt symtim latency)
```

```
            pos=tptr - start + 60/rate*20 + (detect2-23)/rate*10  % relative pos to start
        elseif (detect3)
            start=tptr + (detect3-23-71)*10 - 600 % abs. reference (subt symtim latency)
            pos=tptr - start + 60/rate*20 + (detect3-23)/rate*10  % relative pos to start
        end

        % Currently ignoring syncwords 4-5!!
        if (detect1|detect2|detect3)
            sync=1;
            if (pos<1200+3*20)
                newptr=start+1200+3*20; % absolute reference, add shift to ctr of gt.
            else
                newptr=start+2400+3*20; % too late, missed start of second burst
            end
        else
            tptr=tptr+600;
        end


end


return;
```

PLLHLF.M

```
function
[detect11all,detect21all,nmetric1all,nfreq1all,max1all,CRQI1all,detect12all,detect22all,nmetric2all,nfreq2
all,max2all,CRQI2all,detect13all,detect23all,nmetric3all,nfreq3all,max3all,CRQI3all,newptr]=pllhlf(ifseq
d,atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8)
%
% Parallel implementation of demodulator code.
% Symbol timing implemented one burst vector at a time.
% Coherent recovery still uses serial looping.
% 2nd order carrier recovery . . . "fast" poles
% presumes 10.7 MHz input IF, sampled at 3840 kHz, associated compensations



% ONLY PURPOSE IS TO ACQUIRE SYNC, AFTER A DETECT RESET AND CALL OUT TO FULL
BURST DEMOD

% CORRELATION PATTERNS
% Note: these are NOT carefully chosen, should choose patterns with
% large minimum distance to avoid error aliasing of one pattern
% for another
corrpat1=[0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 1 0 1];  % must don't care symbol 0 (diff'l encoding)
corrpat2=[0 0 1 1 1 0 1 1 0 0 0 0 1 0 0 0 0 0 1 1 0 1];
corrpat3=[0 0 1 1 1 0 1 1 1 0 0 0 1 0 0 0 0 0 1 1 0 1];
corrpat4=[0 0 1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 1];
corrpat5=[0 0 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 0 1 1 0 1];


% SYM TIM CONSTANT VECTORS

invert=1;                    % bit to account for any inversions in IF

% IF compensation to add 140 kHz
% done as 9,9,10,9,9,10 brad per sample! (9,9,9,... would give only 135 kHz)
ifcomp=[10;9;9]*ones(1,200); ifcomp=ifcomp(:);

% deal with wraparound
ifcomp=do2scomp(cumsum(ifcomp)',8);

% filter coefficients (what about Rob's extra latch in bpphase2.m?)
bpfilt=[1,0,-1];
xfilt=[0,1,0,-2,0,1];
yfilt=[0,0,2,0,-2,0];

% make mod 4 counter (vector of 1200)
cnt=modulo(1:600,4);
cnt1=floor(cnt/2);
cnt0=modulo(cnt,2);

start=0;                     % Start symbol of accumulation window in burst
finish=30;                   % Stop symbol of accumulation window in burst

% CARRIER RECOVERY CONSTANTS
```

```
alpha=1/2;              % alpha gain in 2nd order loop
beta=1/16;              % beta gain in 2nd order loop

% if sequence is signed format (not 2s comp)

index=0;
rate=10;  % rate multiplication for fast carrier recovery

%%% PATH 1

tptr1=1;
curr_burst1=zeros(1,600);

% initialize delay variables

% Symbol Timing
stmaxd1=zeros(1,3);
dph8vecd1=zeros(1,30);
nmetricd1=zeros(10,1);
nfreqd1=zeros(10,1);
stPLLd1=zeros(1,4);
gateph8d1=zeros(1,30);

% Carrier Recovery
dout1=zeros(1,30);


%%% PATH 2

tptr2=201;
curr_burst2=zeros(1,600);

% initialize delay variables

% Symbol Timing
stmaxd2=zeros(1,3);
dph8vecd2=zeros(1,30);
nmetricd2=zeros(10,1);
nfreqd2=zeros(10,1);
stPLLd2=zeros(1,4);
gateph8d2=zeros(1,30);

% Carrier Recovery
dout2=zeros(1,30);


%%% PATH 3

tptr3=401;
curr_burst3=zeros(1,600);


% initialize delay variables
```

```
% Symbol Timing
stmaxd3=zeros(1,3);
dph8vecd3=zeros(1,30);
nmetricd3=zeros(10,1);
nfreqd3=zeros(10,1);
stPLLd3=zeros(1,4);
gateph8d3=zeros(1,30);

% Carrier Recovery
dout3=zeros(1,30);


% check input sequence
rl=length(ifseqd);

sync=0;

while ~(sync),

        index=index+1;


%% Receive Path 1

        % take a half burst of data from input stream
        curr_burst1=ifseqd(tptr1:tptr1+599);

        [stmax1,stPLL1,dph8vec1,gateph81,nmetric1,nfreq1,ph8l1]=symtmhlf(bpfilt,xfilt,yfilt,cnt1,cnt0,
atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,curr_burst1);

        [ffvec1,fphasevec1,bfvec1,bphasevec1,ferrvec1,berrvec1,stCRQI1,dout1]=hlfcr(alpha,beta,stPLLd
1,gateph8d1);

        corrdat1=di2bits(dout1);

        % streaming correlator should work across window slices
        if index>1,
                detect11=correl([bits1all((index-2)*60+38:(index-2)*60+60),corrdat1],corrpat1);
                detect21=correl([bits1all((index-2)*60+38:(index-2)*60+60),corrdat1],corrpat2);
                detect31=correl([bits1all((index-2)*60+38:(index-2)*60+60),corrdat1],corrpat3);
                detect41=correl([bits1all((index-2)*60+38:(index-2)*60+60),corrdat1],corrpat4);
                detect51=correl([bits1all((index-2)*60+38:(index-2)*60+60),corrdat1],corrpat5);
        else
                detect11=0;
                detect21=0;
                detect31=0;
                detect41=0;
                detect51=0;
        end

        %TEMP
        ph8lvec1((index-1)*600+1:(index-1)*600+600)=ph8l1;

        %% Compose diagnostic output vectors
```

```
tptr1all(index)=tptr1;
detect11all(index)=detect11;
detect21all(index)=detect21;
detect31all(index)=detect31;
detect41all(index)=detect41;
detect51all(index)=detect51;
bits1all((index-1)*60+1:(index-1)*60+60)=corrdat1;
fphase1all((index-1)*30+1:(index-1)*30+30)=fphasevec1;
ff1all((index-1)*30+1:(index-1)*30+30)=ffvec1;
ferr1all((index-1)*30+1:(index-1)*30+30)=ferrvec1;
bphase1all((index-1)*30+1:(index-1)*30+30)=bphasevec1;
bf1all((index-1)*30+1:(index-1)*30+30)=bfvec1;
berr1all((index-1)*30+1:(index-1)*30+30)=berrvec1;
CRQI1all(index)=stCRQI1;
dph81all((index-1)*30+1:(index-1)*30+30)=dph8vecd1;
gateph81all((index-1)*30+1:(index-1)*30+30)=gateph8d1;
nmetric1all(:,index)=nmetricd1;
nfreq1all(:,index)=nfreqd1;
max1all(index,:)=stmaxd1;


%%%%% Delay elements to represent pipeline latencies follow
% Symbol Timing
stmaxd1=stmax1;
dph8vecd1=dph8vec1;
nmetricd1=nmetric1;
nfreqd1=nfreq1';
stPLLd1=stPLL1;
gateph8d1=gateph81;

% Carrier Recovery
% NOTE: forward and backward frequency, phase and error vectors
% are not delayed, since they are "instantaneous" outputs
% NONE REQUIRED, since assuming it is run fast enough to complete
% in less that half burst, maybe even in 15 symbols...


%% Receive Path 2

% take a half burst of data from input stream
curr_burst2=ifseqd(tptr2:tptr2+599);

[stmax2,stPLL2,dph8vec2,gateph82,nmetric2,nfreq2,ph8l2]=symtmhlf(bpfilt,xfilt,yfilt,cnt1,cnt0,
atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,curr_burst2);

[ffvec2,fphasevec2,bfvec2,bphasevec2,ferrvec2,berrvec2,stCRQI2,dout2]=hlfcr(alpha,beta,stPLLd
2,gateph8d2);

corrdat2=di2bits(dout2);

% streaming correlator should work across window slices
if index>1,
        detect12=correl([bits2all((index-2)*60+38:(index-2)*60+60),corrdat2],corrpat1);
        detect22=correl([bits2all((index-2)*60+38:(index-2)*60+60),corrdat2],corrpat2);
```

```
        detect32=correl([bits2all((index-2)*60+38:(index-2)*60+60),corrdat2],corrpat3);
        detect42=correl([bits2all((index-2)*60+38:(index-2)*60+60),corrdat2],corrpat4);
        detect52=correl([bits2all((index-2)*60+38:(index-2)*60+60),corrdat2],corrpat5);
else
        detect12=0;
        detect22=0;
        detect32=0;
        detect42=0;
        detect52=0;
end

%TEMP
ph8lvec2((index-1)*600+1:(index-1)*600+600)=ph8l2;

%% Compose diagnostic output vectors
tptr2all(index)=tptr2;
detect12all(index)=detect12;
detect22all(index)=detect22;
detect32all(index)=detect32;
detect42all(index)=detect42;
detect52all(index)=detect52;
bits2all((index-1)*60+1:(index-1)*60+60)=corrdat2;
fphase2all((index-1)*30+1:(index-1)*30+30)=fphasevec2;
ff2all((index-1)*30+1:(index-1)*30+30)=ffvec2;
ferr2all((index-1)*30+1:(index-1)*30+30)=ferrvec2;
bphase2all((index-1)*30+1:(index-1)*30+30)=bphasevec2;
bf2all((index-1)*30+1:(index-1)*30+30)=bfvec2;
berr2all((index-1)*30+1:(index-1)*30+30)=berrvec2;
CRQI2all(index)=stCRQI2;
dph82all((index-1)*30+1:(index-1)*30+30)=dph8vecd2;
gateph82all((index-1)*30+1:(index-1)*30+30)=gateph8d2;
nmetric2all(:,index)=nmetricd2;
nfreq2all(:,index)=nfreqd2;
max2all(index,:)=stmaxd2;


%%%%% Delay elements to represent pipeline latencies follow
% Symbol Timing
stmaxd2=stmax2;
dph8vecd2=dph8vec2;
nmetricd2=nmetric2;
nfreqd2=nfreq2';
stPLLd2=stPLL2;
gateph8d2=gateph82;

% Carrier Recovery
% NOTE: forward and backward frequency, phase and error vectors
% are not delayed, since they are "instantaneous" outputs
% NONE REQUIRED, since assuming it is run fast enough to complete
% in less that half burst, maybe even in 15 symbols...


%% Receive Path 3
```

```
curr_burst3=ifseqd(tptr3:tptr3+599);

[stmax3,stPLL3,dph8vec3,gateph83,nmetric3,nfreq3,ph8l3]=symtmhlf(bpfilt,xfilt,yfilt,cnt1,cnt0,
atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,curr_burst3);

[ffvec3,fphasevec3,bfvec3,bphasevec3,ferrvec3,berrvec3,stCRQI3,dout3]=hlfcr(alpha,beta,stPLLd
3,gateph8d3);

corrdat3=di2bits(dout3);

% streaming correlator should work across window slices
if index>1,
        detect13=correl([bits3all((index-2)*60+38:(index-2)*60+60),corrdat3],corrpat1);
        detect23=correl([bits3all((index-2)*60+38:(index-2)*60+60),corrdat3],corrpat2);
        detect33=correl([bits3all((index-2)*60+38:(index-2)*60+60),corrdat3],corrpat3);
        detect43=correl([bits3all((index-2)*60+38:(index-2)*60+60),corrdat3],corrpat4);
        detect53=correl([bits3all((index-2)*60+38:(index-2)*60+60),corrdat3],corrpat5);
else
        detect13=0;
        detect23=0;
        detect33=0;
        detect43=0;
        detect53=0;
end

%TEMP
ph8lvec3((index-1)*600+1:(index-1)*600+600)=ph8l3;

%% Compose diagnostic output vectors
tptr3all(index)=tptr3;
detect13all(index)=detect13;
detect23all(index)=detect23;
detect33all(index)=detect33;
detect43all(index)=detect43;
detect53all(index)=detect53;
bits3all((index-1)*60+1:(index-1)*60+60)=corrdat3;
fphase3all((index-1)*30+1:(index-1)*30+30)=fphasevec3;
ff3all((index-1)*30+1:(index-1)*30+30)=ffvec3;
ferr3all((index-1)*30+1:(index-1)*30+30)=ferrvec3;
bphase3all((index-1)*30+1:(index-1)*30+30)=bphasevec3;
bf3all((index-1)*30+1:(index-1)*30+30)=bfvec3;
berr3all((index-1)*30+1:(index-1)*30+30)=berrvec3;
CRQI3all(index)=stCRQI3;
dph83all((index-1)*30+1:(index-1)*30+30)=dph8vecd3;
gateph83all((index-1)*30+1:(index-1)*30+30)=gateph8d3;
nmetric3all(:,index)=nmetricd3;
nfreq3all(:,index)=nfreqd3;
max3all(index,:)=stmaxd3;


%%%%% Delay elements to represent pipeline latencies follow
% Symbol Timing
stmaxd3=stmax3;
```

```
        dph8vecd3=dph8vec3;
        nmetricd3=nmetric3;
        nfreqd3=nfreq3';
        stPLLd3=stPLL3;
        gateph8d3=gateph83;

        % Carrier Recovery
        % NOTE: forward and backward frequency, phase and error vectors
        % are not delayed, since they are "instantaneous" outputs
        % NONE REQUIRED, since assuming it is run fast enough to complete
        % in less that half burst, maybe even in 15 symbols...

        [detect11, detect21, detect31, detect41, detect51; detect12, detect22, detect32, detect42, detect52;
detect13, detect23, detect33, detect43, detect53]

        % ASSUMPTION detectx3 will occur before a detectx1 on the following window
        if (detect11)
                start=tptr1 + (detect11-23-23)*10 - 600 % abs. reference (subt symtim latency)
                pos=tptr1 - start + 60/rate*20 + (detect11-23)/rate*10  % relative pos to start

        elseif (detect21)
                start=tptr1 + (detect21-23-47)*10 - 600 % abs. reference (subt symtim latency)
                pos=tptr1 - start + 60/rate*20 + (detect21-23)/rate*10  % relative pos to start

%       elseif (detect31)

        elseif (detect12)
                start=tptr2 + (detect12-23-23)*10 - 600 % abs. reference (subt symtim latency)
                pos=tptr2 - start + 60/rate*20 + (detect12-23)/rate*10  % relative pos to start

        elseif (detect22)
                start=tptr2 + (detect22-23-47)*10 - 600 % abs. reference (subt symtim latency)
                pos=tptr2 - start + 60/rate*20 + (detect22-23)/rate*10  % relative pos to start

%       elseif (detect32)

        elseif (detect13)
                start=tptr3 + (detect13-23-23)*10 - 600 % abs. reference (subt symtim latency)
                pos=tptr3 - start + 60/rate*20 + (detect13-23)/rate*10  % relative pos to start

        elseif (detect23)
                start=tptr3 + (detect23-23-47)*10 - 600 % abs. reference (subt symtim latency)
                pos=tptr3 - start + 60/rate*20 + (detect23-23)/rate*10  % relative pos to start

%       elseif (detect33)

        end

        % Currently ignoring syncwords 3-5!!
        if (detect11|detect21|detect12|detect22|detect13|detect23)
                sync=1;
                if (pos<1200+3*20)
                        newptr=start+1200+3*20; % absolute reference, add shift to ctr of gt.
                else
```

```
                    newptr=start+2400+3*20; % too late, missed start of second burst
            end
        else
            tptr1=tptr1+600;
            tptr2=tptr2+600;
            tptr3=tptr3+600;
        end


end;
```

FASTRCV.M

```
function
[weiall,slipall,payall,bitsall,fphaseall,ffall,ferrall,bphaseall,bfall,berrall,CRQIall,dph8all,gateph8all,nmetri
call,nfreqall,maxall,ph8lvec]=fastrcv(ifseqd,newptr,atn8x6,atn10x6,sq6x8,cs4t6x8,sn4t6x8)
%
% Parallel implementation of demodulator code.
% Symbol timing implemented one burst vector at a time.
% Coherent recovery still uses serial looping.
% 2nd order carrier recovery . . . "fast" poles
% presumes 10.7 MHz input IF, sampled at 3840 kHz, associated compensations



% CORRELATION PATTERN

%corrpat=[0 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1];
corrpat=[0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1];  % must don't care symbol 0 (diff'l encoding)


% SYM TIM CONSTANT VECTORS

invert=1;                % bit to account for any inversions in IF

% IF compensation to add 140 kHz
% done as 9,9,10,9,9,10 brad per sample! (9,9,9,... would give only 135 kHz)
ifcomp=[10;9;9]*ones(1,400); ifcomp=ifcomp(:);

% deal with wraparound
ifcomp=do2scomp(cumsum(ifcomp)',8);

% filter coefficients (what about Rob's extra latch in bpphase2.m?)
bpfilt=[1,0,-1];
xfilt=[0,1,0,-2,0,1];
yfilt=[0,0,2,0,-2,0];

% make mod 4 counter (vector of 1200)
cnt=modulo(1:1200,4);
cnt1=floor(cnt/2);
cnt0=modulo(cnt,2);

start=12;                % Start symbol of accumulation window in burst
finish=49;               % Stop symbol of accumulation window in burst


% CARRIER RECOVERY CONSTANTS

alpha=1/2;               % alpha gain in 2nd order loop
beta=1/16;               % beta gain in 2nd order loop



% if sequence is signed format (not 2s comp)
```

```
tptr=newptr;
index=0;

% FAST, variable rate processing implementation
% NO delay variables, latency approx. 1 burst
% for rate=10 actually 1 burst + 3.3 symbols, which is 8.58 us
% into the 25 us inter-burst gap
streordd=zeros(1,30);


% check input sequence
rl=length(ifseqd);


while tptr <= (floor(rl/1200)-1)*1200+1,

        index=index+1;

        % take a burst of data from input stream
        curr_burst=ifseqd(tptr:tptr+1199);

        [stmax,stPLL,dph8vec,gateph8,nmetric,nfreq,ph8l]=symtimfo(bpfilt,xfilt,yfilt,cnt1,cnt0,atn8x6,at
n10x6,sq6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,curr_burst);

        [ffvec,fphasevec,bfvec,bphasevec,ferrvec,berrvec,stCRQI,streord,dout]=carrec(alpha,beta,streordd
,stPLL,gateph8);

        corrdat=di2bits(dout);

        [wei,slip,payload]=fdecode(corrdat);

        %TEMP
        ph8lvec((index-1)*1200+1:(index-1)*1200+1200)=ph8l;

        %% Compose diagnostic output vectors
        tptrall(index)=tptr;
        weiall(index)=wei;
        slipall(index)=slip;
        payall((index-1)*90+1:(index-1)*90+90)=payload;
        bitsall((index-1)*120+1:(index-1)*120+120)=corrdat;
        fphaseall((index-1)*60+1:(index-1)*60+60)=fphasevec;
        ffall((index-1)*60+1:(index-1)*60+60)=ffvec;
        ferrall((index-1)*60+1:(index-1)*60+60)=ferrvec;
        bphaseall((index-1)*60+1:(index-1)*60+60)=bphasevec;
        bfall((index-1)*60+1:(index-1)*60+60)=bfvec;
        berrall((index-1)*60+1:(index-1)*60+60)=berrvec;
        CRQIall(index)=stCRQI;
        dph8all((index-1)*60+1:(index-1)*60+60)=dph8vec;
        gateph8all((index-1)*60+1:(index-1)*60+60)=gateph8;
        nmetricall(:,index)=nmetric;
        nfreqall(:,index)=nfreq';
        maxall(index,:)=stmax;

        % No Delay elements, pipeline latencies less than one burst
```

```
        % except for carrier recovery which is written with implicit latency
        streordd=streord;

        tptr=tptr+1200;

end

return;
```

SYMTIMFO.M

```
function
[stmax,stPLL,dph8vec,gateph8,nmetric,nfreq,ph8l]=symtimfo(bpfilt,xfilt,yfilt,cnt1,cnt0,atn8x6,atn10x6,s
q6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,ifburst);


        stmax=zeros([1 3]);     % symbol timing results nmetric, gate val, freq offset
        stIa=zeros([1 10]);     % symbol timing I acumulator
        stQa=zeros([1 10]);     % symbol timing Q acumulator
        stPLL=zeros([1 4]);     % PLL refs for carrier recov, states are [fwd phase, fwd freq, bwd
phase, bwd freq]
        invp=invert*ones(1,1200);
        invsgn=-1*invert + (~invert);

        % bandpass filtering (conv yields sequence length m+n-1, throw away first 2 samples)
        dcout=conv(ifburst,bpfilt);
        dcout=dcout(3:1202);

        % mix down from IF / image reject low pass filter (again throw away first 4 samples)
        xout=conv(dcout,xfilt);
        yout=conv(dcout,yfilt);
        xout=xout(6:1205);
        yout=yout(6:1205);

        % x,y should be 7-bit 2's comp numbers (vector of 1200), extract sign
        sgnx=(xout<0);
        sgny=(yout<0);

        % simulate atan ROM lookup on xout[5:2] and yout[5:2]
        % extract correct bits
        xrom=floor(abs(xout)/4);
        yrom=floor(abs(yout)/4);

        % atan and quantization to "brads" (256 brads / 2pi rads)
        romout=atn8x6(1+16*yrom+xrom)';

        % this is now a vector of 1200 six-bit unsigned phases in quadrant I
        % calculate the 7th bit
        d6=xor(invp,xor(cnt0,xor(sgnx,sgny)));

        % calculate the MSB
        d7=xor(invp,( (cnt0 & xor((~cnt1),sgnx)) | ((~cnt0) & xor(cnt1,sgny)) ));
        d50=romout;
        invd50=xor(invp,xor(sgnx,sgny)); % inversion mask vector
        d50=invd50.*(63-d50) + (~invd50).*d50; % inversion of ROM out bits
        ph8u=do2scomp(128*d7+64*d6+d50,8);

        % compensate for IF offset
        % nominal 820 kHz (3*3840 kHz - 10700 kHz), but expecting 960 kHz
        % I think there may be a phase inversion. AGB 7/24/96
        ph8l=do2scomp(ph8u+invsgn*ifcomp,8);

        % DECIMATE ph8 to 10x sampling
```

```
% parity must be 0 or 1, decimate even or odd samples
ph8=ph8l(2:2:1200);

% update input to x & y accumulation in symbol timing
% NOTE another do2scomp for case diff phase > 127 ?
% pad dph8 to 600 samples for consistency, with 10 leading zeroes
dph8(1:10)=zeros([1 10]);
dph8(11:600)=do2scomp(ph8(11:600)-ph8(1:590),8);

dph6=floor(dph8/4); % truncation of 2 LSBs

% calculate ROM address from dph6 (map negative dph6 to 2scomp)
sc4addr=(dph6<0).*(64+dph6) +  (~(dph6<0)).*dph6; % note this is an address VECTOR

% simulate sin4theta and cos4theta lookup ROMs
c4=cs4t6x8(1+sc4addr);  % cos4theta lookup picks vals at address VECTOR indices
s4=sn4t6x8(1+sc4addr);  % sin4theta lookup picks vals at address VECTOR indices

% 14 bit signed accumulation (-8192 to + 8191)
% use zerov to accum middle of burst

for zzz=1:10,
        stIa(zzz)=sum(do2scomp(c4((zzz+start*10):10:finish*10), 8));
        stQa(zzz)=sum(do2scomp(s4((zzz+start*10):10:finish*10), 8));
end

Itrun7=floor(stIa/128); % ASSUMES 14 -> 7 bits (six magnitude)
Qtrun7=floor(stQa/128);
Isqaddr=abs(Itrun7); % I channel address into squaring ROM; NOTE: ASSUMES 6
Qsqaddr=abs(Qtrun7); % Q channel address into squaring ROM; UNSIGNED BITS IN

zzz=find(Isqaddr==64);
Isqaddr(zzz)=63*ones(size(zzz));

nmetric=sq6x8(1+Isqaddr)+sq6x8(1+Qsqaddr);
% this is a 10 element vector, one nmetric for each gate position

stmax(1)=max(nmetric);
maxindex=find(nmetric==max(nmetric)) - 1; % gate val from 0 to 9
%% what about special case for non-unique max?  Arbitrary choice.
stmax(2)=maxindex(1);

Itrun6=floor(stIa/256); % ASSUMES 14 -> 6 BITS (5 magnitude)
Qtrun6=floor(stQa/256); % result is -32 to + 31
Iatn4addr=abs(Itrun6); % I channel address into atan/4 ROM
Qatn4addr=abs(Qtrun6); % Q channel address into atan/4 ROM
sgnI=(Itrun6<0);
sgnQ=(Qtrun6<0);
q50=atn10x6(32*Qatn4addr+Iatn4addr+1)';
q6=xor(sgnI,sgnQ);
q7=sgnQ;
invq50=xor(sgnI,sgnQ);
q50=invq50.*(63-q50) + (~invq50).*q50; % inversion of ROM out bits
q70=do2scomp(128*q7+64*q6+q50,8);
```

```
% this is now the phase 4theta.  Need to subtract pi to get the
% remainder left over due to frequency offset!!!  RZ 4/8/96
nfreq=do2scomp(q70-128,8);

stmax(3)=nfreq(stmax(2)+1);

% diagnostic gated dph8 vector
dph8vec=dph8((stmax(2)+1):10:600);

% gate phase sequence
gateph8=ph8((stmax(2)+1):10:600);

% update PLL phase and frequency refs
% states are [fwd phase, fwd freq, bwd phase, bwd freq]
stPLL([2 4])=do2scomp(16*[stmax(3) stmax(3)],14);
stPLL([1 3])=[0 0];

return;
```

CARREC.M

```
function
[ffvec,fphasevec,bfvec,bphasevec,ferrvec,berrvec,stCRQI,streord,dout]=carrec(alpha,beta,streordin,initPL
L,gateph8);

stPLL=initPLL;
sthburst=zeros([1 30]);     % half-burst RAM, input: phderot, # states: 30
streord=streordin;          % reorder RAM, inputs: {b,f}phaser{2}msb, # states: 30
stCRQI=0;                   % carrier recovery quality index accumulator
stddec=0;                   % diff'l decode, input: dibit, # states: 1

derot=0;


for crcnt=0:59,

        %% COHERENT RECOVERY LOOPS
        %% input from stgate2 which is delayed version of properly gated (stmax(2)) _absolute_ phase
ph8
        %% what about reorder ram,  streord and half burst ram, sthburst
        %% registers: stPLL, remember to initialize
        %% counters: crcnt, (actually increments on tcbaud (symbol rate), cleared by tcburst, ie: 0 to 59)
        %% flags: derot=32*modulo(crcnt,2);

        %% implement as for loop on crcnt, ignore tcbaud

        % derotate the incoming phase
        phderot=do2scomp(gateph8(crcnt+1)+derot,8);

        % forward and backward errors
        % remember phderot is phase used
        % make the errors 6 bits!
        ferr=do2scomp(phderot-floor(stPLL(1)/64),6);
        if crcnt < 30,
                bphderot=phderot;
        else,
                bphderot=256*(sthburst(59-crcnt+1) < 0) + sthburst(59-crcnt+1);
        end
        berr=do2scomp(bphderot-floor(stPLL(3)/64),6);

        % phase and frequency register inputs:
        % poles at (2-alpha)/2 (critically damped beta=alpha^2/4)
        bphasein=do2scomp(stPLL(3)+stPLL(4)+alpha*64*berr,14);
        bfin=do2scomp(stPLL(4)+beta*64*berr,14);
        fphasein=do2scomp(stPLL(1)+stPLL(2)+alpha*64*ferr,14);
        ffin=do2scomp(stPLL(2)+beta*64*ferr,14);

        % combinatorial inputs to reorder RAM
        % are written in 2nd half of burst
        bphaseout=do2scomp(bphderot-floor(stPLL(3)/64)+32,8);
        fphaseout=do2scomp(phderot-floor(stPLL(1)/64)+32,8);
        fphaser=256*(fphaseout<0)+fphaseout;
        fphasermsb=floor(fphaser/128);
```

```
fphaser2msb=floor((fphaser-128*fphasermsb)/64);
bphaser=256*(bphaseout<0)+bphaseout;
bphasermsb=floor(bphaser/128);
bphaser2msb=floor((bphaser-128*bphasermsb)/64);

% current dibit output from reorder RAM
if crcnt < 30,
        tmsb=floor(streord(29-crcnt+1)/8);
        tlsb=floor((streord(29-crcnt+1)-8*tmsb)/4);
        creord=streord(29-crcnt+1);
else,
        tmsb=floor(modulo(streord(crcnt-30+1),4)/2);
        tlsb=modulo(streord(crcnt-30+1),2);
        creord=streord(crcnt-30+1);
end
dibit=2*tmsb+tlsb;

% dibits of past symbol "dibit" (registered in stddec)
dmsb=floor(stddec/2);
dlsb=modulo(stddec,2);

b1=xor(dmsb,dlsb);
b0=dlsb;
a1=tmsb;
a0=tlsb;
vout=modulo(2*b1+b0+2*a1+a0,4);
omsb=floor(vout/2);
olsb=modulo(vout,2);

zmsb=xor(omsb,( (~olsb) & derot) );
zlsb=xor(omsb,( (~derot) & olsb) );

dout(crcnt+1)=2*zmsb+zlsb;

% Carrier Recovery Quality Index
% (temporarily uses explicit square operation)
crqiin=ferr^2+berr^2;

% diagnostic
fphasevec(crcnt+1)=stPLL(1);
ffvec(crcnt+1)=stPLL(2);
ferrvec(crcnt+1)=ferr;
bphasevec(crcnt+1)=stPLL(3);
bfvec(crcnt+1)=stPLL(4);
berrvec(crcnt+1)=berr;

%%%%%%%%% (symbol) clock rising edge; update the registers

if (crcnt==29),
        % tie forward and backward together
        % BACKWARD frequency loop
        % grab the forward guys at count 29
        stPLL(4)=do2scomp(-stPLL(2),14);
```

```
            % no change to stPLL(3)

            % still need FORWARD phase loop
            stPLL(1)=fphasein;

            % FORWARD frequency loop
            stPLL(2)=ffin;

        else,
            % BACKWARD phase loop
            stPLL(3)=bphasein;

            % BACKWARD frequency loop
            stPLL(4)=bfin;

            % FORWARD phase loop
            stPLL(1)=fphasein;

            % FORWARD frequency loop
            stPLL(2)=ffin;
        end

        if (crcnt >= 30),
            % reorder RAM for output bits
            % complete reorder RAM: backward bits, forward bits
            streord(crcnt-30+1)=[bphasermsb bphaser2msb fphasermsb fphaser2msb]*[8;4;2;1];
        end

        % update diff'l decode register
        stddec=dibit;

        % half-burst RAM for fwd/bwd carrier recovery
        if (crcnt < 30),
            sthburst(1+crcnt)=phderot;
        end

        % Carrier Recovery Quality Index register
        if (crcnt == 0),
            stCRQI=0;
        end
        if ((crcnt>29) & (crcnt<46)),
            stCRQI=stCRQI+crqiin;
        end

        derot=32*modulo(crcnt,2);

        %%%%%%%%%% register updates done

        %%% END COHERENT RECOVERY CODE

end

return;
```

SYMTMHLF.M

```
function
[stmax,stPLL,dph8vec,gateph8,nmetric,nfreq,ph8l]=symtmhlf(bpfilt,xfilt,yfilt,cnt1,cnt0,atn8x6,atn10x6,s
q6x8,cs4t6x8,sn4t6x8,start,finish,invert,ifcomp,ifburst);


        stmax=zeros([1 3]);     % symbol timing results nmetric, gate val, freq offset
        stIa=zeros([1 10]);     % symbol timing I acumulator
        stQa=zeros([1 10]);     % symbol timing Q acumulator
        stPLL=zeros([1 4]);     % PLL refs for carrier recov, states are [fwd phase, fwd freq, bwd
phase, bwd freq]
        invp=invert*ones(1,600);
        invsgn=-1*invert + (~invert);

        % bandpass filtering (conv yields sequence length m+n-1, throw away first 2 samples)
        dcout=conv(ifburst,bpfilt);
        dcout=dcout(3:602);

        % mix down from IF / image reject low pass filter (again throw away first 4 samples)
        xout=conv(dcout,xfilt);
        yout=conv(dcout,yfilt);
        xout=xout(6:605);
        yout=yout(6:605);

        % x,y should be 7-bit 2's comp numbers (vector of 600), extract sign
        sgnx=(xout<0);
        sgny=(yout<0);

        % simulate atan ROM lookup on xout[5:2] and yout[5:2]
        % extract correct bits
        xrom=floor(abs(xout)/4);
        yrom=floor(abs(yout)/4);

        % atan and quantization to "brads" (256 brads / 2pi rads)
        romout=atn8x6(1+16*yrom+xrom)';

        % this is now a vector of 600 six-bit unsigned phases in quadrant I
        % calculate the 7th bit
        d6=xor(invp,xor(cnt0,xor(sgnx,sgny)));

        % calculate the MSB
        d7=xor(invp,( (cnt0 & xor((~cnt1),sgnx)) | ((~cnt0) & xor(cnt1,sgny)) ));
        d50=romout;
        invd50=xor(invp,xor(sgnx,sgny)); % inversion mask vector
        d50=invd50.*(63-d50) + (~invd50).*d50; % inversion of ROM out bits
        ph8u=do2scomp(128*d7+64*d6+d50,8);

        % compensate for IF offset
        % nominal 820 kHz (3*3840 kHz - 10700 kHz), but expecting 960 kHz
        % I think there may be a phase inversion. AGB 7/24/96
        ph8l=do2scomp(ph8u+invsgn*ifcomp,8);

        % DECIMATE ph8 to 10x sampling
```

```
% parity must be 0 or 1, decimate even or odd samples
ph8=ph8l(2:2:600);

% update input to x & y accumulation in symbol timing
% NOTE another do2scomp for case diff phase > 127 ?
% pad dph8 to 300 samples for consistency, with 10 leading zeroes
dph8(1:10)=zeros([1 10]);
dph8(11:300)=do2scomp(ph8(11:300)-ph8(1:290),8);

dph6=floor(dph8/4); % truncation of 2 LSBs

% calculate ROM address from dph6 (map negative dph6 to 2scomp)
sc4addr=(dph6<0).*(64+dph6) +  (~(dph6<0)).*dph6; % note this is an address VECTOR

% simulate sin4theta and cos4theta lookup ROMs
c4=cs4t6x8(1+sc4addr);  % cos4theta lookup picks vals at address VECTOR indices
s4=sn4t6x8(1+sc4addr);  % sin4theta lookup picks vals at address VECTOR indices

% 14 bit signed accumulation (-8192 to + 8191)
% use zerov to accum middle of burst

for zzz=1:10,
        stIa(zzz)=sum(do2scomp(c4((zzz+start*10):10:finish*10), 8));
        stQa(zzz)=sum(do2scomp(s4((zzz+start*10):10:finish*10), 8));
end

Itrun7=floor(stIa/128); % ASSUMES 14 -> 7 bits (six magnitude)
Qtrun7=floor(stQa/128);
Isqaddr=abs(Itrun7); % I channel address into squaring ROM; NOTE: ASSUMES 6
Qsqaddr=abs(Qtrun7); % Q channel address into squaring ROM; UNSIGNED BITS IN

zzz=find(Isqaddr==64);
Isqaddr(zzz)=63*ones(size(zzz));

nmetric=sq6x8(1+Isqaddr)+sq6x8(1+Qsqaddr);
% this is a 10 element vector, one nmetric for each gate position

stmax(1)=max(nmetric);
maxindex=find(nmetric==max(nmetric)) - 1; % gate val from 0 to 9
%% what about special case for non-unique max?  Arbitrary choice.
stmax(2)=maxindex(1);

Itrun6=floor(stIa/256); % ASSUMES 14 -> 6 BITS (5 magnitude)
Qtrun6=floor(stQa/256); % result is -32 to + 31
Iatn4addr=abs(Itrun6); % I channel address into atan/4 ROM
Qatn4addr=abs(Qtrun6); % Q channel address into atan/4 ROM
sgnI=(Itrun6<0);
sgnQ=(Qtrun6<0);
q50=atn10x6(32*Qatn4addr+Iatn4addr+1)';
q6=xor(sgnI,sgnQ);
q7=sgnQ;
invq50=xor(sgnI,sgnQ);
q50=invq50.*(63-q50) + (~invq50).*q50; % inversion of ROM out bits
q70=do2scomp(128*q7+64*q6+q50,8);
```

```
% this is now the phase 4theta.  Need to subtract pi to get the
% remainder left over due to frequency offset!!!  RZ 4/8/96
nfreq=do2scomp(q70-128,8);

stmax(3)=nfreq(stmax(2)+1);

% diagnostic gated dph8 vector
dph8vec=dph8((stmax(2)+1):10:300);

% gate phase sequence
gateph8=ph8((stmax(2)+1):10:300);

% update PLL phase and frequency refs
% states are [fwd phase, fwd freq, bwd phase, bwd freq]
stPLL([2 4])=do2scomp(16*[stmax(3) stmax(3)],14);
stPLL([1 3])=[0 0];

return;
```

HLFCR.M

```
function
[ffvec,fphasevec,bfvec,bphasevec,ferrvec,berrvec,stCRQI,dout]=carrec(alpha,beta,initPLL,gateph8);

% NOTE: This is all implemented in one function for use in
% half burst latency loop in hlfrcv.m,  Depending on rate
% multiplier over symbol rate, could have latency as low as
% 15 symbols => correlation as early as 45 symbols after air,


stPLL=initPLL;
sthburst=zeros([1 30]);     % half-burst RAM, input: phderot, # states: 30
streord=zeros([1 30]);      % reorder RAM, inputs: {b,f}phaser{2}msb, # states: 30
stCRQI=0;                   % carrier recovery quality index accumulator
stddec=0;                   % diff'l decode, input: dibit, # states: 1

derot=0;

%% COHERENT RECOVERY LOOPS
%% implement as for loop on crcnt, ignore tcbaud

for crcnt=0:29,

        % derotate the incoming phase
        phderot=do2scomp(gateph8(crcnt+1)+derot,8);

        % forward and backward errors
        % remember phderot is phase used
        % make the errors 6 bits!
        ferr=do2scomp(phderot-floor(stPLL(1)/64),6);

        % phase and frequency register inputs:
        % poles at (2-alpha)/2 (critically damped beta=alpha^2/4)
        fphasein=do2scomp(stPLL(1)+stPLL(2)+alpha*64*ferr,14);
        ffin=do2scomp(stPLL(2)+beta*64*ferr,14);

        % diagnostic
        fphasevec(crcnt+1)=stPLL(1);
        ffvec(crcnt+1)=stPLL(2);
        ferrvec(crcnt+1)=ferr;

        %%%%%%%%% (symbol) clock rising edge; update the registers

        % FORWARD phase loop
        stPLL(1)=fphasein;

        % FORWARD frequency loop
        stPLL(2)=ffin;

        % half-burst RAM for fwd/bwd carrier recovery
        sthburst(1+crcnt)=phderot;

        derot=32*modulo(crcnt,2);
```

```
        %%%%%%%%%% register updates done

        %%% END COHERENT RECOVERY CODE FORWARD HALF
end


% Initialize backward loop freq and phase registers
% (could reuse forward loop registers, but instead
% keeping structure of full burst carrier recovery)
stPLL(4)=do2scomp(-stPLL(2),14);
stPLL(3)=stPLL(1);
stCRQI=0;

for crcnt=30:59,

        bphderot=256*(sthburst(59-crcnt+1) < 0) + sthburst(59-crcnt+1);
        berr=do2scomp(bphderot-floor(stPLL(3)/64),6);

        % phase and frequency register inputs:
        % poles at (2-alpha)/2 (critically damped beta=alpha^2/4)
        bphasein=do2scomp(stPLL(3)+stPLL(4)+alpha*64*berr,14);
        bfin=do2scomp(stPLL(4)+beta*64*berr,14);

        % combinatorial inputs to reorder RAM
        % are written in 2nd half of burst
        bphaseout=do2scomp(bphderot-floor(stPLL(3)/64)+32,8);
        bphaser=256*(bphaseout<0)+bphaseout;
        bphasermsb=floor(bphaser/128);
        bphaser2msb=floor((bphaser-128*bphasermsb)/64);

        % Carrier Recovery Quality Index
        % (temporarily uses explicit square operation)
        crqiin=berr^2;

        % diagnostic
        bphasevec(crcnt-29)=stPLL(3);
        bfvec(crcnt-29)=stPLL(4);
        berrvec(crcnt-29)=berr;

        %%%%%%%%% (symbol) clock rising edge; update the registers

        % BACKWARD phase loop
        stPLL(3)=bphasein;

        % BACKWARD frequency loop
        stPLL(4)=bfin;

        % reorder RAM for output bits
        % complete reorder RAM: backward bits, forward bits
        streord(crcnt-30+1)=[bphasermsb bphaser2msb]*[8;4];

        % Carrier Recovery Quality Index register
        if ((crcnt>29) & (crcnt<45)),
```

```
            stCRQI=stCRQI+crqiin;
        end

        %%%%%%%%%% register updates done

        %%% END COHERENT RECOVERY CODE BACKWARD HALF

end


derot=0;

%% Differential decode of reorder ram output (very low latency)

for crcnt=0:29,

        % current dibit output from reorder RAM
        tmsb=floor(streord(29-crcnt+1)/8);
        tlsb=floor((streord(29-crcnt+1)-8*tmsb)/4);

        dibit=2*tmsb+tlsb;

        % dibits of past symbol "dibit" (registered in stddec)
        dmsb=floor(stddec/2);
        dlsb=modulo(stddec,2);

        b1=xor(dmsb,dlsb);
        b0=dlsb;
        a1=tmsb;
        a0=tlsb;
        vout=modulo(2*b1+b0+2*a1+a0,4);
        omsb=floor(vout/2);
        olsb=modulo(vout,2);

        zmsb=xor(omsb,( (~olsb) & derot) );
        zlsb=xor(omsb,( (~derot) & olsb) );

        dout(crcnt+1)=2*zmsb+zlsb;

        %%%%%%%%% (symbol) clock rising edge; update the registers

        % update diff'l decode register
        stddec=dibit;

        derot=32*modulo(crcnt,2);

        %%%%%%%%%% register updates done

        %%% END COHERENT RECOVERY CODE DIFFL DECODE

end

return;
```

CORREL.M

```
function detect = correl(source, pattern);

% NOTE: match is 1 at index of END of found pattern
%       this is done to resemble a shift reg correlator implementation

match=zeros(size(source));

for i=1:length(source)-length(pattern)+1
        matchvec=source(i:i+length(pattern)-1)==pattern;
        match(i+length(pattern)-1)=~(length(find(matchvec==0)));
end

detect=find(match==1);
if (length(detect) > 0),
        detect=detect(1);
else
        detect=0;
end
```

FDECODE.M

```
function [wei, slip, dpayload] = fdecode(bits);

% ASSUME CENTER OF GUARD TIME IS NOMINALLY AT INDEX 1 OF bits VECTOR

slips=-6:2:6;                   % vector of slip offsets
ptr=1:120:length(bits);
wei=0;
slip=0;
dpayload=0;

for j=1:length(ptr),
        for i=1:7,
                cw=bits(ptr(j)+slips(i)+8:ptr(j)+slips(i)+8+104);       % skip 6 bits gt + 2 bits diff encode
                weivec(i)=errchk(cw);
        end;
        wei(j)=~(length(find(weivec==0))==1);
        if wei(j)==0,
                slip(j)=slips(find(weivec==0));
                dpayload((j-1)*90+1:(j-1)*90+90)=bits(ptr(j)+slip(j)+8:ptr(j)+slip(j)+8+89);
                dpayload((j-1)*90+1)=~(dpayload((j-1)*90+1)); % unmark first marker bit (second is in
CRC)
        else
                slip(j)=0;
                dpayload((j-1)*90+1:(j-1)*90+90)=zeros([1 90]);
        end;
end
```

ERRCHK.M

```
function wei=errchk(cw);
%
% does error check using PACS channel code given
% codeword (slow and fast channel bits and crc bits)
%
% codeword given as 105 bit vector; [bit0 bit1 ... bit104]
% codeword is divided and remainder is checked to be zero


% unmark
cw(1)=~cw(1);
cw(105)=~cw(105);

% do long division
dividend=cw;

divisor=[1,1,1,1,1,0,1,1,0,1,0,0,0,0,0,1]; % PACS standard generator

done=0;

while ~done,
        % determine the position for the quotient
        % pos=k -> dividend has term at x^(105-k)
        pos=min(find(dividend==1));
        if pos <= 90,
                % subtract the appropriate multiple of the divisor
                sterm=[zeros(1,pos-1),divisor,zeros(1,90-pos)];
                dividend=modulo(dividend+sterm,2);
        else
                % we're done!
                done=1;
        end     % end if
end     % end while

% now what is left of the dividend should be zero
% (the remainder)
wei=length(find(~(dividend==0)))>0;
return;
```

ATAN10X6.M

```
% atan10x6.m
%
% calculates entries for 1024 word x 6 bits ROM for
% arctan lookup in demodulator logic design
% ASSUMES final divide-by-four is done in logic to get Foffset!!!

% input address bits a[9..0] map to [y4 y3 y2 y1 y0 x4 x3 x2 x1 x0]
% output data bits d[5..0] map to an angle in the first quadrant
% with resolution 45/32 degrees per "tick"

% generate address list
a=(0:1023)'; abits=zeros(1024,8); abits=dec2bin(a,10);

% calculate first-quadrant arctangent of y[4..0]/x[4..0]
yarg=abits(:,1:5)*[16;8;4;2;1];
xarg=abits(:,6:10)*[16;8;4;2;1];

% calculate the arctangent and convert to appropriate integer units
atn10x6=floor(128*atan(yarg ./ xarg)/pi);

% deal with the 90 degree case; assume atan(0/0) = 90 degrees
% but since 90 degrees = "64" and we have 6 bits, we must truncate
% at 63!
zzz=find(xarg==0);
atn10x6(zzz)=63*ones(length(zzz),1);

% now output angle numbers . . .

% first the MAT file
save atan10x6.mat atn10x6

end;
```

ATAN8X6.M

```
% atan8x6.m
%
% calculates entries for 256 word x 6 bits ROM for
% arctan lookup in demodulator logic design

% input address bits a[7..0] map to [y3 y2 y1 y0 x3 x2 x1 x0]
% output data bits d[5..0] map to an angle in the first quadrant
% with resolution 45/32 degrees per "tick"

% generate address list
a=(0:255)'; abits=zeros(256,8); abits=dec2bin(a,8);

% calculate first-quadrant arctangent of y[3..0]/x[3..0]
yarg=abits(:,1:4)*[8;4;2;1];
xarg=abits(:,5:8)*[8;4;2;1];

% calculate the arctangent and convert to appropriate integer units
atn8x6=floor(128*atan(yarg ./ xarg)/pi);

% deal with the 90 degree case; assume atan(0/0) = 90 degrees
% but since 90 degrees = "64" and we have 6 bits, we must truncate
% at 63!
zzz=find(xarg==0);
atn8x6(zzz)=63*ones(length(zzz),1);

% now output angle numbers . . .

% first the MAT file
save atan8x6.mat atn8x6

end;
```

COS4T6X8.M

```
% cos4t6x8.m
%
% calculates entries for 64 word x 8 bits ROM for
% cos(4theta) lookup in demodulator logic design

% input address bits a[5..0] represent a signed twos complement angle
% output data bits d[7..0] represent a signed twos complement result
% that scales the cos(4theta) by 128

% generate address list
a=(0:63)'; abits=zeros(64,8); abits=dec2bin(a,6);

% convert twos comp binary representation to signed angle
% remember we took the upper 6 bits of what was an 8 bit signed angle
angs=-1*abits(:,1) .* (32*ones(64,1)-abits(:,2:6)*[16;8;4;2;1]) + ...
    (ones(64,1)-abits(:,1)) .* (abits(:,2:6)*[16;8;4;2;1]);
% convert units to radians
angs=pi*angs/32;

% calculate the cosine and convert to appropriate integer units
cs4t6x8=round(128*cos(4*angs));

% deal with the 4theta= 0, 2*pi, etc. case.  There cos(4theta)=+1
% which scales to +128, but we have 8 bits, so we must truncate
% at +127!
zzz=find(cs4t6x8==128);
cs4t6x8(zzz)=127*ones(length(zzz),1);

% now convert signed decimal integers to 8 bit signed twos comp!
zzz=find(cs4t6x8 < 0);
cs4t6x8(zzz)=256*ones(length(zzz),1)+cs4t6x8(zzz);

% now output angle numbers . . .

% first the MAT file
save cos4t6x8.mat cs4t6x8

end;
```

SIN4T6X8.M

```
% sin4t6x8.m
%
% calculates entries for 64 word x 8 bits ROM for
% sin(4theta) lookup in demodulator logic design

% input address bits a[5..0] represent a signed twos complement angle
% output data bits d[7..0] represent a signed twos complement result
% that scales the sin(4theta) by 128

% generate address list
a=(0:63)'; abits=zeros(64,8); abits=dec2bin(a,6);

% convert twos comp binary representation to signed angle
% remember we took the upper 6 bits of what was an 8 bit signed angle
angs=-1*abits(:,1) .* (32*ones(64,1)-abits(:,2:6)*[16;8;4;2;1]) + ...
    (ones(64,1)-abits(:,1)) .* (abits(:,2:6)*[16;8;4;2;1]);
% convert units to radians
angs=pi*angs/32;

% calculate the sine and convert to appropriate integer units
sn4t6x8=round(128*sin(4*angs));

% deal with the 4theta= 0, 2*pi, etc. case.  There sin(4theta)=+1
% which scales to +128, but we have 8 bits, so we must truncate
% at +127!
zzz=find(sn4t6x8==128);
sn4t6x8(zzz)=127*ones(length(zzz),1);

% now convert signed decimal integers to 8 bit signed twos comp!
zzz=find(sn4t6x8 < 0);
sn4t6x8(zzz)=256*ones(length(zzz),1)+sn4t6x8(zzz);

% now output angle numbers . . .

% first the MAT file
save sin4t6x8.mat sn4t6x8

end;
```

SQR6X8.M

```
% sqr6x8.m
%
% calculates entries for 64 word x 8 bits ROM for
% squaring lookup in demodulator logic design

% input address bits a[5..0] represent an unsigned number
% output data bits d[7..0] represent an unsigned result, scaled
% by 1/16 (since 6 bits x 6 bits <= 12 bits)

a=(0:63)';

% square a, divide by 16, take nearest integer
sq6x8=round(a .* a / 16);

% now output numbers . . .
% first the MAT file
save sqr6x8.mat sq6x8

end;
```

DI2BITS.M

```
function bits=di2bits(dibits);

bits=0;
lobits=modulo(dibits,2);
hibits=floor(dibits/2);
bits(1:2:length(lobits)*2)=hibits;
bits(2:2:length(lobits)*2)=lobits;

return;
```

DO2SCOMP.M

```
function out=do2scomp(in,n)
%
% do2scomp.m
%
% takes n bit phase sequence and turns into n-bit 2s comp sequence
% with values -2^(n-1) <= x <= 2^(n-1) - 1

% if not a row, MAKE IT ONE!
in=in(:)';

% get size of input
lin=length(in);

% first get in [0,2^n)
out=modulo(in,2^n);

% now take the upper half of the range and pull it back
% pull [2^(n-1),2^n) back to [-2^(n-1),0)
zzz=find(out > 2^(n-1)-1); lzzz=length(zzz);
out(zzz)=out(zzz)-2^n * ones(1,lzzz);

return;
```

MODULO.M

```
function [out]=modulo(in,M);
%
% takes value of input modulo M
% output: answer
% input: input value, M (make M integer)

nrevs=floor(in/M);
out=in - nrevs*M;
```