

Validation of 802.11a/UWB Coexistence Simulation  
L. E. Miller, 10/17/03

1. TRANSMITTED WAVEFORMS.....	2
1.1 802.11a OFDM Signal.....	2
1.2 UWB Interference Signal.....	6
1.2.1 General model.....	6
1.2.2 Example using $M$ -ary orthogonal keying.....	8
2. RECEIVED WAVEFORMS .....	9
2.1 802.11a OFDM Signal.....	9
2.2 Receiver Noise Waveform.....	11
2.3 UWB Baseband Waveform.....	12
2.3.1 Effect at baseband of a single UWB pulse.....	13
2.3.2 Effect at baseband of a train of modulated UWB pulses .....	16
2.3.3 Samples and FFT of the UWB interference.....	17
3. PREDICTED PERFORMANCE .....	19
3.1 Performance of OFDM in Noise Only.....	19
3.1.1 Probability of bit error without coding .....	19
3.1.2 Probability of bit error with coding .....	23
3.1.2.1 BPSK and QPSK.....	23
3.1.2.2 QAM .....	24
3.2 Performance of OFDM in Noise and UWB Interference .....	30
3.2.1 UWB without multipath.....	30
3.2.2 UWB with multipath.....	32
4. VALIDATION OF SIMULATION MODEL .....	32
4.1 Performance in the AWGN Channel .....	32
4.2 Performance with UWB Interference .....	35
APPENDICES	
A. Mapping of Binary Data to Modulation Symbols.....	38
B. Calculation of Received OFDM Channel Noise Powers .....	40
C. Simulation Data.....	41
D. Simulation Program .....	44
REFERENCES .....	75

## 1. TRANSMITTED WAVEFORMS

### 1.1 802.11a OFDM Signal

Let  $a(t)$  denote the sequence of complex baseband (I/Q) data modulation symbols generated by the data source, with or without error-control coding prior to the mapping of the binary data to modulation symbols. Details of the mapping are given in Appendix A. The symbol rate is fixed at  $1/T = 12$  megasymbols per second (Msps). Each symbol  $a_k = a(kT)$  represents  $K$  (coder output) bits according the modulation used (BPSK, QPSK, 16-QAM, or 64-QAM), so that the effective data rate in Mbps equals  $12Kr$ , where  $r$  is the code rate ( $1/2$ ,  $2/3$ , or  $3/4$ ).

For transmission using orthogonal frequency division multiplexing (OFDM) according to the IEEE 802.11a standard, the data modulation symbols are normalized, buffered in groups of 48 symbols, and then transmitted in parallel on different subcarriers at the OFDM symbol rate of  $12 \text{ Msps}/48 = 250 \text{ Ksps} = 1/48T = 1/T_s$ . Let the vector  $\mathbf{a}_n$  denote the  $n$ th set of 48 modulation symbols:

$$\begin{aligned}\mathbf{a}_n &= \left\{ a[(n-1)T_s]/\kappa, a[(n-1)T_s + T]/\kappa, \dots, a[(n-1)T_s + 47T]/\kappa \right\} \\ &= \{\mathbf{a}_{n0}, \mathbf{a}_{n1}, \dots, \mathbf{a}_{n47}\}\end{aligned}\quad (1.1.1a)$$

where the normalization factor  $\kappa$  is based on the fact that

$$\langle |a_k|^2 \rangle = \begin{cases} 1, & \text{BPSK} \\ 2, & \text{QPSK} \\ 10, & \text{16-QAM} \\ 42, & \text{64-QAM} \end{cases} \triangleq \kappa^2 \quad (1.1.1b)$$

To facilitate signal acquisition and channel estimation, in 802.11a, pilot signals are interspersed with the data-modulated carriers, as illustrated in Figure 1.1.1, in which the carrier spacing is  $\Delta_F = 20 \text{ MHz}/64 = 0.3125 \text{ MHz}$ , resulting in a bandwidth of  $53\Delta_F = 16.5625 \text{ MHz}$ . The pilot signals are modulated at the OFDM symbol rate by a  $\pm 1$  pseudorandom noise (PN) sequence. This allocation of carriers to data and pilots can be represented by inserting a zero and a PN sequence value  $p$  into  $\mathbf{a}_n$  in five places to create a 53-sample vector,  $\mathbf{b}_n$ , as follows:

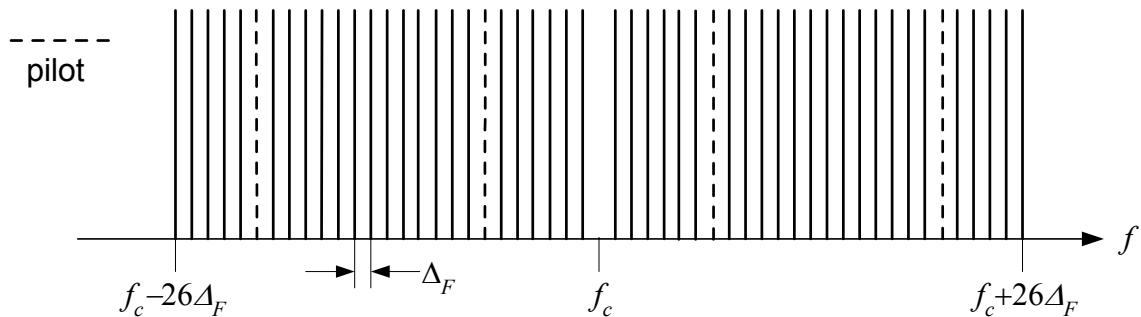


Figure 1.1.1 Sketch of 802.11a OFDM spectrum.

$$\begin{aligned}\mathbf{b}_n &= \{\mathbf{a}_{n0}, \dots, \mathbf{a}_{n4}, p, \mathbf{a}_{n5}, \dots, \mathbf{a}_{n17}, p, \mathbf{a}_{n18}, \dots, \mathbf{a}_{n23}, 0, \mathbf{a}_{n24}, \dots, \mathbf{a}_{n29}, p, \mathbf{a}_{n30}, \dots, \mathbf{a}_{n42}, -p, \mathbf{a}_{n43}, \dots, \mathbf{a}_{n47}\} \\ &= \{\mathbf{b}_{n,-26}, \mathbf{b}_{n,-25}, \dots, \mathbf{b}_{n,25}, \mathbf{b}_{n,26}\}\end{aligned}\quad (1.1.2)$$

Allowing for division of the OFDM symbol period into an initial guard time ( $t_g$ ) and a symbol transmission time ( $T_s - t_g$ ), during the  $T_s = 4 \mu\text{s}$  OFDM symbol period the quadrature components of the complex baseband OFDM signal for the  $n$ th OFDM symbol have the form

$$s_n(t) = I_n(t) + jQ_n(t) = \sum_{k=-26}^{26} b_{nk} e^{j2\pi k \Delta_F (t-t_g)} \quad (1.1.3a)$$

or

$$I_n(t) = \operatorname{Re} \left\{ \sum_{k=-26}^{26} b_{nk} e^{j2\pi k \Delta_F (t-t_g)} \right\} = \sum_{k=-26}^{26} |b_{nk}| \cos [2\pi k \Delta_F (t-t_g) + \varphi_{nk}] \quad (1.1.3b)$$

and

$$Q_n(t) = \operatorname{Im} \left\{ \sum_{k=-26}^{26} b_{nk} e^{j2\pi k \Delta_F (t-t_g)} \right\} = \sum_{k=-26}^{26} |b_{nk}| \sin [2\pi k \Delta_F (t-t_g) + \varphi_{nk}] \quad (1.1.3c)$$

where

$$\varphi_{nk} = \tan^{-1} \left\{ \operatorname{Im}[b_{nk}] / \operatorname{Re}[b_{nk}] \right\} \quad (1.1.3d)$$

The quantities in (1.1.3b) and (1.1.3c) can be multiplied by a window function to accomplish pulse waveshaping for minimizing the sidelobes of the signal spectrum. After insertion of an RF carrier (upbanding) and power amplification to obtain a power of  $P_{Tc}$  per carrier, the OFDM waveform during the  $n$ th symbol period has the form

$$\begin{aligned}s_{Tn}(t) &= \operatorname{Re} \left\{ \sqrt{2P_{Tc}} (I_n + jQ_n) e^{-j2\pi f_c t + j\theta} \right\} = \sqrt{2P_{Tc}} [I_n(t) \cos(2\pi f_c t + \theta) + Q_n(t) \sin(2\pi f_c t + \theta)] \\ &= \sqrt{2P_{Tc}} \sum_{k=-26}^{26} |b_{nk}| \cos [2\pi (f_c + k \Delta_F) (t-t_g) + \varphi_{nk} + \theta']\end{aligned}\quad (1.1.4)$$

where  $\theta$  and  $\theta'$  are carrier phases that are fixed over the duration of the symbol. The transmitted power of the signal in (1.1.4), averaged over different OFDM symbols and over possible symbol data values, is given by

$$P_T = P_{Tc} \left\langle \sum_{k=-26}^{26} |b_{nk}|^2 \right\rangle = P_{Tc} (52 \left\langle |b_{nk}|^2 \right\rangle) = 52P_{Tc} \quad (1.1.5)$$

The modulation of the subcarriers by the data modulation symbols is equivalent to using an inverse discrete Fourier transform (IDFT). Recall [1] that the discrete Fourier transform (DFT) of  $N$  samples of a time series  $\{x_m = x(m\Delta_T), m = 0, \dots, N-1\}$  is the set of  $N$  spectral coefficients given by

$$X_k = X(k \cdot 2\pi\Delta_F) = \Delta_T \sum_{m=0}^{N-1} x_m e^{-j2\pi(k\Delta_F)(m\Delta_T)} = \Delta_T \sum_{m=0}^{N-1} x_m e^{-j2\pi km/N}, \quad k = 0, \dots, N-1 \quad (1.1.6a)$$

where  $X(\omega)$  is the Fourier transform of the periodic extension of the waveform  $x(t)$  over the interval  $t = 0$  to  $t = T_0 = N\Delta_T$ . Note that the DFT is periodic— $X_{-k} = X_{N-k}$ —and that the frequency spacing of the spectral coefficients equals  $\Delta_F = 1/T_0 = 1/N\Delta_T$ . Similarly, the (periodic) time series can be recovered by the IDFT, which is given by

$$x_m = x(m \cdot \Delta_T) = \Delta_F \sum_{k=0}^{N-1} X_k e^{j2\pi km/N} = \Delta_F \sum_{k=-N/2+1}^{N/2} X_k e^{j2\pi km/N}, \quad m = 0, \dots, N-1 \quad (1.1.6b)$$

Samples of the waveform for the  $n$ th OFDM symbol in (1.1.3a) have the form of the IDFT in (1.1.6b) for some value of  $\Delta_T$ :

$$s_n(t_g + m\Delta_T) = \sum_{k=-26}^{26} b_{nk} e^{j2\pi(k\Delta_F)(m\Delta_T)} \quad (1.1.7a)$$

The 802.11a standard uses a symbol guard time of  $t_g = 800$  ns, which leave a symbol transmission time of  $T_s - t_g = 3.2$   $\mu$ s. For efficient calculation of the IDFT using a fast Fourier transform (FFT), a 64-point IDFT can be used, based on selecting  $T_0 = 3.2$   $\mu$ s,  $\Delta_F = 1/T_0 = 312.5$  KHz and  $\Delta_T = 3.2$   $\mu$ s/64 = 50 ns. For these choices, (1.1.7a) can be manipulated as follows:

$$s_{nm} = s_n(t_g + m\Delta_T) = \sum_{k=-31}^{32} c_{nk} e^{j2\pi km/64} = \sum_{k=0}^{63} c_{nk} e^{j2\pi km/64} \quad (1.1.7b)$$

where the periodic sequence  $\mathbf{c}_n$  is formed by appending zeros to the beginning and end of  $\mathbf{b}_n$  to obtain

$$\mathbf{c}_n = \{c_{n,-31}, c_{n,-30}, \dots, c_{n,31}, c_{n,32}\} = \{0, 0, 0, 0, 0, b_{n,-26}, \dots, b_{n,26}, 0, 0, 0, 0\} \quad (1.1.8)$$

Note that the value of  $s_n(t)$  during the guard time preceding the OFDM symbol is the periodic extension of the samples generated using (1.1.8), for a total of 80 samples.

If a smaller sample time than 50 ns is desired, then a larger FFT size than 64 can be used. The resulting sample times are shown in Table 1.1.1.

The average power in the baseband OFDM symbol during the 3.2  $\mu$ s OFDM symbol transmission time is found to be

Table 1.1.1 Sample time vs. FFT size

FFT size	128	256	512	1024	2048	4096	8192
$\Delta_T$	25 ns	12.5 ns	6.25 ns	3.125 ns	1.5625 ns	0.78125 ns	0.390625 ns

$$\begin{aligned}
P_{BB} &= \frac{1}{64} \left\langle \sum_{m=0}^{63} |s_{nm}|^2 \right\rangle = \frac{1}{64} \left\langle \sum_{m=0}^{63} \left| \sum_{k=-26}^{26} b_{nk} e^{j2\pi km/64} \right|^2 \right\rangle \\
&= \frac{1}{64} \left\langle \sum_{r=-26}^{26} \bar{b}_{nr} \sum_{k=-26}^{26} b_{nk} \sum_{m=0}^{63} e^{j2\pi(k-r)m/64} \right\rangle = \left\langle \sum_{k=-26}^{26} |b_{nk}|^2 \right\rangle = 52
\end{aligned} \tag{1.1.9a}$$

This result is based on the fact that

$$\sum_{m=0}^{63} e^{j2\pi(k-r)m/64} = 64\delta_{kr} = \begin{cases} 64, & k = r \\ 0, & k \neq r \end{cases} \tag{1.1.9b}$$

Alternatively, the power can be measured over the entire 4  $\mu$ s OFDM symbol interval, including the guard time at the beginning of the interval. The expected average power using these 80 samples is the same, as shown by the following:

$$\begin{aligned}
P_{BB} &= \frac{1}{80} \left\langle \sum_{m=-16}^{63} |s_{nm}|^2 \right\rangle = \frac{64}{80} \left\langle \sum_{k=-26}^{26} |b_{nk}|^2 \right\rangle + \frac{1}{80} \sum_{m=-16}^{-1} \left[ \sum_{k=-26}^{26} \left\langle |b_{nk}|^2 \right\rangle + \sum_{r \neq k} \left\langle b_{nk} \bar{b}_{nr} \right\rangle e^{j2\pi(k-r)m/64} \right] \\
&= \frac{64}{80} \left\langle \sum_{k=-26}^{26} |b_{nk}|^2 \right\rangle + \frac{16}{80} \left\langle \sum_{k=-26}^{26} |b_{nk}|^2 \right\rangle = \left\langle \sum_{k=-26}^{26} |b_{nk}|^2 \right\rangle = 52
\end{aligned} \tag{1.1.10a}$$

This result is based on (1.1.9b) and on the fact that the modulation symbols are independent and zero-mean:

$$\left\langle b_{nk} \bar{b}_{nr} \right\rangle = \begin{cases} \left\langle |b_{nk}|^2 \right\rangle, & k = r \\ 0, & k \neq r \end{cases} \tag{1.1.10b}$$

The transmitted bit energy of the OFDM symbol, based on a bit rate of  $12Kr$  MHz, is given by

$$E_{bT} = \frac{P_T}{R_b} = \frac{52P_{Tc}}{12Kr} 10^{-6} = \frac{13P_{Tc}}{3Kr} 10^{-6} \tag{1.1.11}$$

## 1.2 UWB Interference Signal

### 1.2.1 General model

The interfering UWB signal is modeled at its transmitter as

$$z(t) = C_T \sum_{l=-\infty}^{\infty} u_l g(t - lT_u) \quad (1.2.1a)$$

where the data  $\{u_l\}$  are assumed to be independent random variables that are equally likely to take values of +1 or -1. An example of the pulse  $g(t)$  is an integer number of cycles of a sinusoid:

$$g(t) = \begin{cases} \sin(\omega_r t), & 0 \leq t \leq N_u T_r \\ 0, & \text{otherwise} \end{cases}, \quad \omega_r = \frac{2\pi}{T_r} \quad (1.2.1.b)$$

The Fourier transform of this pulse is given by [3]

$$G(\omega) = F\{g(t)\} = \left(1 - e^{-j\omega N_u T_r}\right) \frac{1/\omega_r}{1 - (\omega/\omega_r)^2} = e^{-j\omega N_u T_r / 2} \sin\left(\frac{\omega N_u T_r}{2}\right) \frac{2j/\omega_r}{1 - (\omega/\omega_r)^2} \quad (1.2.2)$$

The data signal  $z(t)$  is cyclostationary and therefore has the autocorrelation function [4]

$$R_z(\tau) = \frac{1}{T_u} \int_{-T_u/2}^{T_u/2} E\{z(t)z(t+\tau)\} dt = \frac{C_T^2}{T_u} \int_{-T_u/2}^{T_u/2} g(t)g(t+\tau) dt = \frac{C_T^2}{T_u} \cdot g(t)*g(-t) \quad (1.2.3a)$$

assuming the duration of the pulse is smaller than the pulse repetition period ( $N_u T_r \ll T_u$ ). Thus the average power of  $z(t)$  is

$$P_{T_u} = R_z(0) = \frac{C_T^2}{T_u} \int_0^{N_u T_r} \sin^2(\omega_r t) dt = \frac{C_T^2}{2} \cdot \frac{N_u T_r}{T_u} \quad (1.2.3b)$$

which can be interpreted as the average power of a sinusoid times the duty cycle of the pulsed waveform.

As illustrated in Figure 1.2.1 for the case of  $T_r = 0.25$  ns, the spectrum of  $z(t)$  for the  $N_u$ -cycle sinusoidal pulse is

$$|Z(\omega)|^2 = \frac{C_T^2}{T_u} |G(\omega)|^2 = \frac{4C_T^2}{\omega_r^2 T_u} \cdot \frac{\sin^2(\omega N_u T_r / 2)}{\left[1 - (\omega/\omega_r)^2\right]^2} \quad (1.2.4)$$

Note how the spectrum gradually becomes concentrated about the frequency  $f_r = 1/T_r$  as the number of cycles increases. The extremely wide bandwidth of the UWB signal can be appreciated from Figure 1.2.2, in which the example spectrum of the UWB signal is shown in comparison to two hypothetical 802.11a signals at the lowest and highest center frequencies.

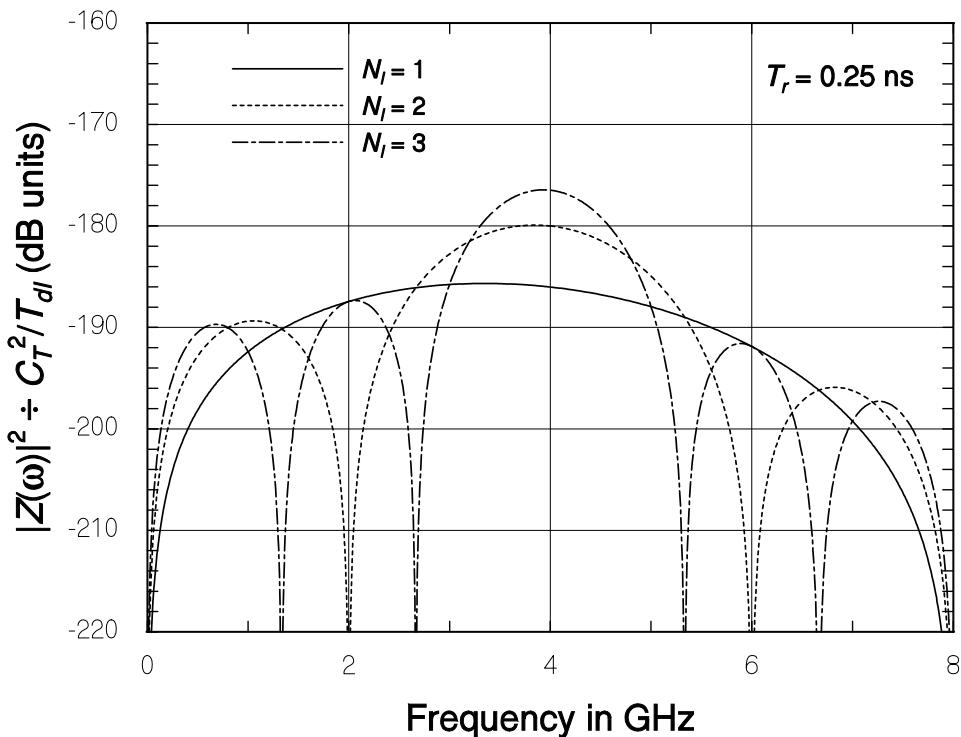


Figure 1.2.1 Example spectrum of UWB signal.

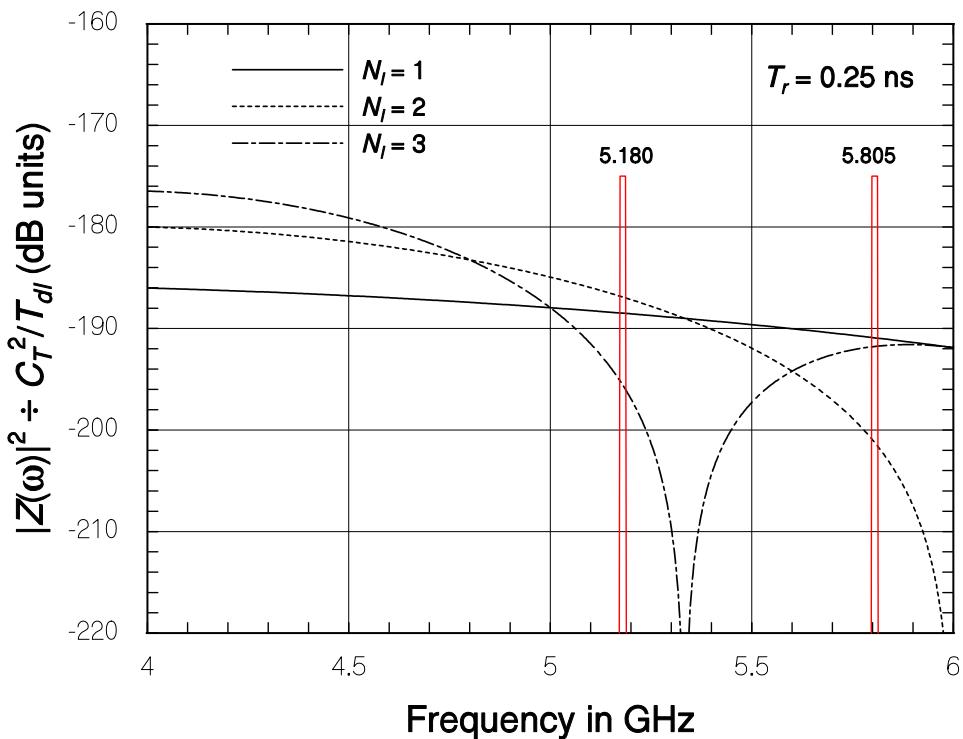


Figure 1.2.2 Comparison of UWB and 802.11a spectrums

Note than if the data values  $\{u_l\}$  are not random and zero-mean, there may be a periodic component of the UWB signal, leading to lines in its spectrum. An extreme example is the case in which the signal is an unmodulated pulse train ( $u_l = 1$  for all  $l$ ); for this case, in Woodward's notation [8, 15], the UWB signal is

$$z(t) = C_T \sum_{l=-\infty}^{\infty} g(t - lT_u) = C_T \text{Rep}_{T_u}\{g(t)\} \quad (1.2.5)$$

for which the Fourier transform is [8]

$$Z(\omega) = \frac{1}{T_u} \sum_{n=-\infty}^{\infty} G(2\pi n/T_u) \delta(\omega - 2\pi n/T_u) = \frac{1}{T_u} \text{Comb}_{2\pi/T_u}\{G(\omega)\} \quad (1.2.6)$$

The spectrum based on (1.2.6) is a line spectrum with line spacing  $1/T_u$ . Depending on the pulse rate, one or more lines will fall within the passband of the 802.11a receiver.

### 1.2.2 Example using $M$ -ary orthogonal keying

An example data waveform for a pulsed UWB signal is  $M$ -ary orthogonal keying, in which  $K = \log_2 M$  bits select one of  $M$  orthogonal sequences, such as Walsh sequences, transmitted by means of  $M$  signed pulses. Each sequence of pulses is equivalent to an  $M$ -ary symbol.

For example, groups of  $K = 4$  bits select one of  $M = 2^K = 16$  Walsh sequences consisting of 16 pulses with special combinations of positive and negative amplitudes. Given the bit rate,  $R_b$ , for this example the pulse rate equals  $(16/4) R_b$  and the symbol rate equals  $(1/4) R_b$ . The four bits of data, the binary equivalent of decimal 6, selects the Walsh sequence with six zero crossings [8], which determines the polarity of the sixteen pulses that encode the four bits, as illustrated in Figure 1.2.3. A drawback to this scheme is the fact that the first digit of each Walsh sequence is a zero, so there would be a repeating pattern of a logical zero at the beginning of each symbol period.

Using the same approach to modulation,  $K + 1$  bits can be encoded using  $M = 2^K$  orthogonal sequences by the simple expedient of using one bit to manipulate the polarity of the entire sequence in what is referred to as a bi-orthogonal signaling scheme [4] known as  $M$ -ary bi-orthogonal keying (MBOK). This scheme has the additional advantage that, on the average, every pulse is equally likely to have a positive or a negative amplitude. Additional, pseudo-random coding can be added to provide isolation for UWB sub-nets.

The highest value of the pulse rate  $(1/T_u)$  is determined by the duration of the pulse  $(N_u T_r)$ , which is approximately the inverse of the bandwidth. Setting  $T_u = N_u T_r$ , the highest symbol rate for an MBOK scheme equals  $1/MN_u T_r$  and the highest bit rate equals  $(K + 1)/MN_u T_r$ .

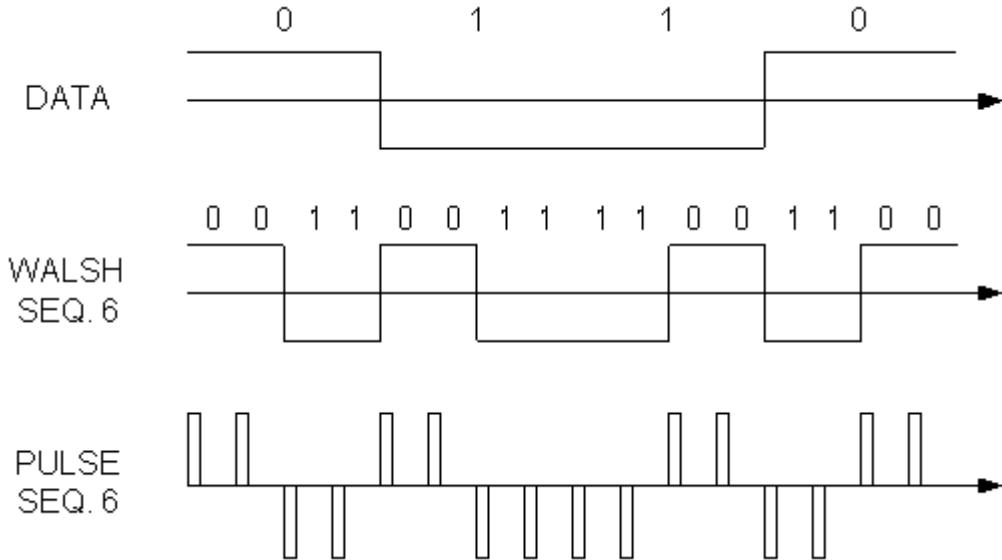


Figure 1.2.3 Example 16-ary orthogonal keying UWB pulse waveforms

## 2. RECEIVED WAVEFORMS

The received RF waveform during reception of the  $n$ th OFDM symbol is modeled as

$$r(t) = s_{Rn}(t) + z_R(t) + v(t) \quad (2.1)$$

where  $s_{Rn}(t)$  denotes the OFDM signal,  $z_R(t)$  denotes the interfering UWB signal, and  $v(t)$  denotes receiver noise.

### 2.1 802.11a OFDM Received Waveform

During reception of the  $n$ th OFDM symbol, the received OFDM waveform is given by

$$s_{Rn}(t) = \sqrt{2P_{Rc}} \sum_{k=-26}^{26} |b_{nk}| \cos[2\pi(f_c + k\Delta_F)(t - t_g) + \varphi_{nk} + \theta''] \quad (2.1.1)$$

Using  $h_0(t)$  to denote the lowpass filter impulse response in the I/Q receiver, assuming local oscillators with effective amplitudes of  $A = 2$ , and assuming that receiver synchronization and carrier acquisition are performed perfectly, the recovered quadrature OFDM waveforms are

$$\begin{aligned} I_{Rn}(t) &= h_0(t) * [s_{Rn}(t) \cdot 2 \cos(2\pi f_c t + \theta'')] \approx [s_{Rn}(t) \cdot 2 \cos(2\pi f_c t + \theta'')]_{\text{Lowpass}} \\ &= \sqrt{2P_{Rc}} \sum_{k=-26}^{26} |b_{nk}| \cos[2\pi k \Delta_F (t - t_g) + \varphi_{nk}] \end{aligned} \quad (2.1.2a)$$

and

$$\begin{aligned}
Q_{Rn}(t) &= -h_0(t) * [s_{Rn}(t) \cdot 2 \sin(2\pi f_c t + \theta'')] \approx -[s_{Rn}(t) \cdot 2 \sin(2\pi f_c t + \theta'')]_{\text{Lowpass}} \\
&\approx \sqrt{2P_{Rc}} \sum_{k=-26}^{26} |b_{nk}| \sin \left[ 2\pi k \Delta_F (t - t_g) + \varphi_{nk} \right]
\end{aligned} \tag{2.1.2b}$$

In practice, the received power per OFDM carrier  $P_{Rc}$  reflects any front-end gains in the receiver implementation. Samples of the recovered I and Q components of the OFDM symbol following the guard time form the  $N_p$  complex samples given by

$$\begin{aligned}
s_{Rnm} &= I_{Rn}(t_g + m\Delta_T) + jQ_{Rn}(t_g + m\Delta_T) = \sqrt{2P_{Rc}} \sum_{k=-26}^{26} |b_{nk}| e^{j2\pi(k\Delta_F)(m\Delta_T) + j\varphi_{nk}} \\
&= \sqrt{2P_{Rc}} \sum_{k=-26}^{26} b_{nk} e^{j2\pi(k\Delta_F)(m\Delta_T)} = \sqrt{2P_{Rc}} \sum_{k=0}^{N_p-1} c_{nk} e^{j2\pi km/N_p}
\end{aligned} \tag{2.1.3}$$

The  $i$ th output of an FFT of these samples equals

$$\begin{aligned}
S_{Rni} &= \frac{1}{N_p} \sum_{m=0}^{N_p-1} s_{Rnm} e^{-j2\pi mi/N_p} = \frac{\sqrt{2P_{Rc}}}{N_p} \sum_{m=0}^{N_p-1} \sum_{k=0}^{N_p-1} c_{nk} e^{j2\pi m(k-i)/N_p} \\
&= \frac{\sqrt{2P_{Rc}}}{N_p} \sum_{k=0}^{N_p-1} c_{nk} \sum_{m=0}^{N_p-1} e^{j2\pi m(k-i)/N_p} = \sqrt{2P_{Rc}} \sum_{k=0}^{N_p-1} c_{nk} \delta_{ki} = \sqrt{2P_{Rc}} \cdot c_{ni}
\end{aligned} \tag{2.1.4a}$$

or

$$S_{Rni} = \sqrt{2P_{Rc}} c_{ni} = \begin{cases} \sqrt{2P_{Rc}} b_{ni}, & i = 1, \dots, 26 \\ \sqrt{2P_{Rc}} b_{n,-k}, & i = N_p - k, k = 1, \dots, 26 \\ 0, & \text{otherwise} \end{cases} \tag{2.1.4b}$$

Thus, in the absence of noise and interference, an FFT of the input I/Q samples yields an array containing the original complex data modulation symbols. The average bit energy in one of the nonzero FFT data outputs is

$$E_{bR} = \frac{P_R}{R_b} = \frac{52P_{Rc}}{12Kr} 10^{-6} = \frac{13P_{Rc}}{3Kr} 10^{-6} \tag{2.1.5}$$

where  $P_R$  is the total received power in the non-guard time portion of the OFDM symbol and  $P_{Rc}$  is the received power on one of the OFDM carriers.

## 2.2 Receiver Noise Waveform

The receiver noise referred to the receiver input may be modeled as additive white Gaussian noise over some bandwidth  $W$  that is much larger than the receiver RF bandwidth,  $B$ . Let the two-sided power spectral density of the noise be denoted  $N_0/2$ . Then the Gaussian noise waveform at the receiver input in its bandwidth,  $B$ , can be modeled as

$$v(t) = v_c(t)\cos(2\pi f_c t) - v_s(t)\sin(2\pi f_c t) \quad (2.2.1)$$

where the quadrature noise components  $v_c(t)$  and  $v_s(t)$  are independent, zero-mean Gaussian random processes. For an ideal (rectangular) receiver filter, each quadrature component has the autocorrelation function

$$R_v(\tau) = N_0 \int_{-B/2}^{B/2} df e^{j2\pi f\tau} = N_0 B \cdot \sin(\pi B\tau)/\pi B\tau = N_0 B \text{sinc}(B\tau) \quad (2.2.2)$$

Note that (2.2.2) implies that samples of the quadrature noise components taken at intervals of  $\Delta t = 1/B$  are uncorrelated. Again assuming perfect synchronization and carrier acquisition, the noise components in the receiver's quadrature outputs are, respectively,  $v_c(t)$  and  $v_s(t)$ .

Let samples of the noise quadrature components during the  $n$ th OFDM symbol period be denoted by  $v_c(m\Delta_T) = v_{cm}$  and  $v_s(m\Delta_T) = v_{sm}$ . Note that the sample rate is higher than the input noise bandwidth. These samples are processed by the FFT in the OFDM receiver to yield complex output noise components at the  $i$ th frequency, denoted  $\mathcal{N}_{ni} = \mathcal{N}_{nil} + j\mathcal{N}_{niQ}$ , that are given by

$$\mathcal{N}_{ni} = \frac{1}{N_p} \sum_{m=0}^{N_p-1} (v_{mc} + jv_{ms}) e^{-j2\pi mi/N_p} \quad (2.2.3)$$

These output components are independent, zero-mean Gaussian random variables with variances given by  $\sigma_{ni}^2 = E\{\mathcal{N}_{nil}^2\} = E\{\mathcal{N}_{niQ}^2\}$ , where it is shown in Appendix B that

$$\sigma_{ni}^2 \doteq N_0 \Delta_F \times \begin{cases} 1 - \frac{2}{\pi^2} \frac{53}{(53)^2 - (2i)^2}, & |i| \leq 26 \\ -\frac{2}{\pi^2} \frac{53}{(53)^2 - (2i)^2}, & 26 < |i| \leq 32 \end{cases} \quad (2.2.4)$$

with the convention that  $i = 0$  at the center of the band. The variation in the noise power with the index is graphed in Figure 2.2.1. Note how the oversampling of the noise affects the value of noise power near the edges of the signal band.

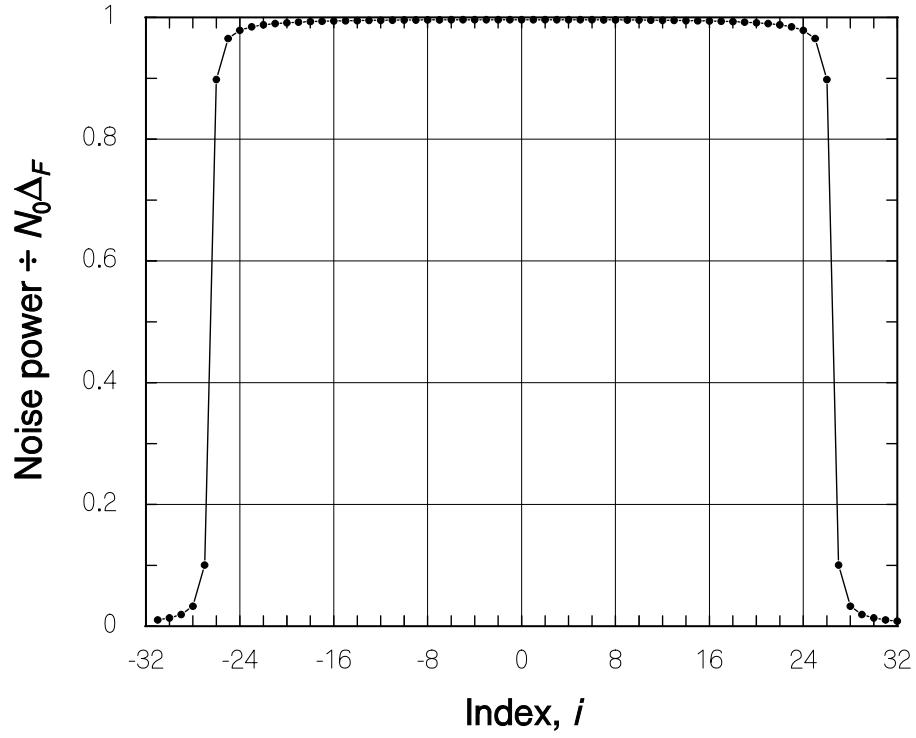


Figure 2.2.1 Noise power (variance) in OFDM receiver FFT output bins.

### 2.3 UWB Baseband Waveform

For an IQ baseband receiver, the incoming signal is subject to zonal bandpass filtering then I and Q quadrature heterodyning to baseband using lowpass filters with impulse response  $h_0(t)$  to extract the I and Q components of the signal, respectively. Thus, using the asterisk (\*) to denote convolution,

$$I(t) = [r(t) \times \cos(\omega_c t + \varphi)] * h_0(t), \quad Q(t) = [r(t) \times (-\sin(\omega_c t + \varphi))] * h_0(t) \quad (2.3.1)$$

in which  $\varphi$  denotes a random initial phase for the local oscillator. In the frequency domain, the Fourier transforms of the quadrature components are

$$I(\omega) = \frac{1}{2} [e^{j\varphi} R(\omega - \omega_c) + e^{-j\varphi} R(\omega + \omega_c)] H_0(\omega) \quad (2.3.2a)$$

and

$$Q(\omega) = \frac{j}{2} [e^{j\varphi} R(\omega - \omega_c) - e^{-j\varphi} R(\omega + \omega_c)] H_0(\omega) \quad (2.3.2b)$$

where  $R(\omega)$  is the Fourier transform of the received waveform. First, we find the effect at baseband for a single UWB pulse; afterwards, we find the effect of a train of modulated UWB pulses.

### 2.3.1 Effect at baseband of a single UWB pulse

Taking the inverse Fourier transform of (2.3.2a), we have the in-phase quadrature component of the received UWB pulse at baseband given by

$$p_I(t) = \frac{1}{2\pi} \int_{-\pi B}^{\pi B} d\omega e^{j\omega t} \left\{ \frac{1}{2} [e^{j\varphi} G(\omega - \omega_c) + e^{-j\varphi} G(\omega + \omega_c)] H_0(\omega) \right\} \quad (2.3.3)$$

where the range of integration is determined by the baseband filter bandwidth,  $B$ . For  $B \ll \omega_c / 2\pi$ , the expression in the brackets is practically equal to its value for  $\omega = 0$ , so the in-phase quadrature component evaluates to

$$\begin{aligned} p_I(t) &\doteq \frac{1}{2} [e^{j\varphi} G(-\omega_c) + e^{-j\varphi} G(\omega_c)] \cdot \frac{1}{2\pi} \int_{-\pi B}^{\pi B} d\omega e^{j\omega t} H_0(\omega) \\ &= \frac{1}{2} [e^{j\varphi} G(-\omega_c) + e^{-j\varphi} G(\omega_c)] h_0(t) \end{aligned} \quad (2.3.4)$$

Substituting (1.2.2) in (2.3.4), we find for the  $N_u$ -cycle sinusoidal pulse that

$$\begin{aligned} p_I(t) &\doteq \frac{1}{2} \left[ e^{j\varphi + jN_u \omega_c T_r / 2} \frac{2j \sin(-N_u \omega_c T_r / 2) / \omega_r}{1 - (\omega_c / \omega_r)^2} + e^{-j\varphi - jN_u \omega_c T_r / 2} \frac{2j \sin(N_u \omega_c T_r / 2) / \omega_r}{1 - (\omega_c / \omega_r)^2} \right] h_0(t) \\ &= \frac{2 \sin(N_u \omega_c T_r / 2 + \varphi) \sin(N_u \omega_c T_r / 2)}{\omega_r [1 - (\omega_c / \omega_r)^2]} h_0(t) \end{aligned} \quad (2.3.5)$$

Similarly, based on (2.3.2b), the cross-quadrature component of the baseband UWB pulse is formulated as

$$p_Q(t) = \frac{1}{2\pi} \int_{-\pi B}^{\pi B} d\omega e^{j\omega t} \left\{ \frac{j}{2} [e^{j\varphi} G(\omega - \omega_c) - e^{-j\varphi} G(\omega + \omega_c)] H_0(\omega) \right\} \quad (2.3.6)$$

For  $B \ll \omega_c / 2\pi$ , the expression in the brackets is practically equal to its value for  $\omega = 0$ , so the cross-quadrature component evaluates to

$$\begin{aligned} p_Q(t) &\doteq \frac{j}{2} [e^{j\varphi} G(-\omega_c) - e^{-j\varphi} G(\omega_c)] \cdot \frac{1}{2\pi} \int_{-\pi B}^{\pi B} d\omega e^{j\omega t} H_0(\omega) \\ &= \frac{j}{2} [e^{j\varphi} G(-\omega_c) - e^{-j\varphi} G(\omega_c)] h_0(t) \end{aligned} \quad (2.3.7)$$

Substituting (1.2.2) in (2.3.7), we find that

$$\begin{aligned} p_Q(t) &\doteq \frac{j}{2} \left[ e^{j\varphi + jN_u \omega_c T_r / 2} \frac{2j \sin(-N_u \omega_c T_r / 2) / \omega_r}{1 - (\omega_c / \omega_r)^2} - e^{-j\varphi - jN_u \omega_c T_r / 2} \frac{2j \sin(N_u \omega_c T_r / 2) / \omega_r}{1 - (\omega_c / \omega_r)^2} \right] h_0(t) \\ &= \frac{2 \cos(N_u \omega_c T_r / 2 + \varphi) \sin(N_u \omega_c T_r / 2)}{\omega_r [1 - (\omega_c / \omega_r)^2]} h_0(t) \end{aligned} \quad (2.3.8)$$

Evaluated for the example numerical values of  $1/T_r = 4\text{GHz}$  and  $f_c = 5\text{ GHz}$ , we obtain the following quadrature components, of which the in-phase component is plotted in Figure 2.3.1, using for illustration purposes  $1/T_d = 20\text{MHz} \approx B$  to denote the baseband signal data symbol duration:

$$p_I(t) \approx -(1.4 \times 10^{-10}) \sin(5N_u \pi / 4 + \varphi) \sin(5N_u \pi / 4) h_0(t) \quad (2.3.9a)$$

and

$$p_Q(t) \approx -(1.4 \times 10^{-10}) \cos(5N_u \pi / 4 + \varphi) \sin(5N_u \pi / 4) h_0(t) \quad (2.3.9b)$$

Another numerical example is shown in Figure 2.3.2. In this example, a UWB sinusoidal burst with a Gaussian-shaped envelope [3] has a spectrum centered at 4.5 GHz, while the narrowband signal is at 5 GHz with an assumed bandwidth of 16 MHz. Intuitively, the larger the bandwidth of the UWB signal, the larger the effect on its neighbor at 5 GHz, and also the more closely the UWB spectrum appears to be flat in the band of that signal. However, an interesting effect of the downconversion is that an image of the signal causes the effective spectrum of the

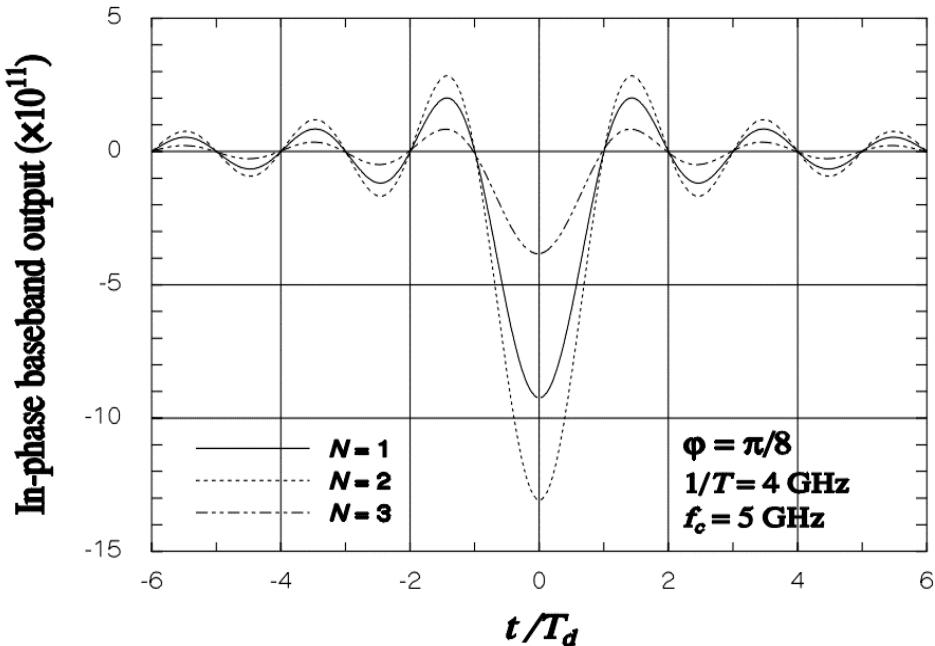


Figure 2.3.1  $p_I(t)$  for numerical example

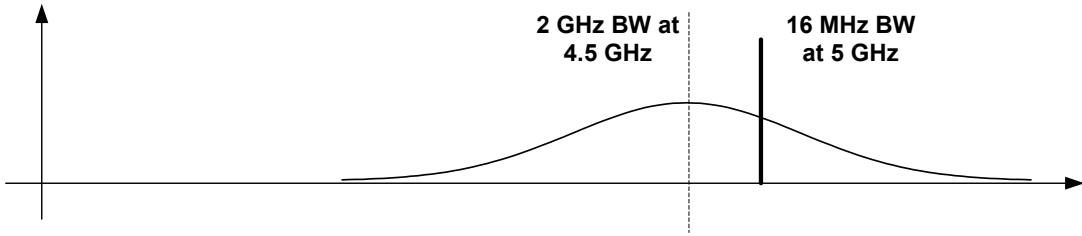


Figure 2.3.2 Example of UWB interference to a narrowband receiver.

UWB interferer to have even symmetry in the receiver baseband, making the approximation described above work for cases in which only the edge of the UWB spectrum at RF interferes with the narrowband signal. Figure 2.3.3 illustrates how the downconverted signal and its image overlap at the receiver baseband. The effect of UWB bandwidth on the interference in the receiver baseband is illustrated in Figure 2.3.4 for the example Gaussian-envelope sinusoidal burst; note that the overlap of the downconverted UWB spectrum and its image causes the baseband interference spectrum to be even and tends to flatten the interference until the UWB signal is practically out-of-band and extremely weak. Figure 2.3.5 shows that the response of the baseband filter to the UWB interference is very close to that predicted above, until the interference is very weak.

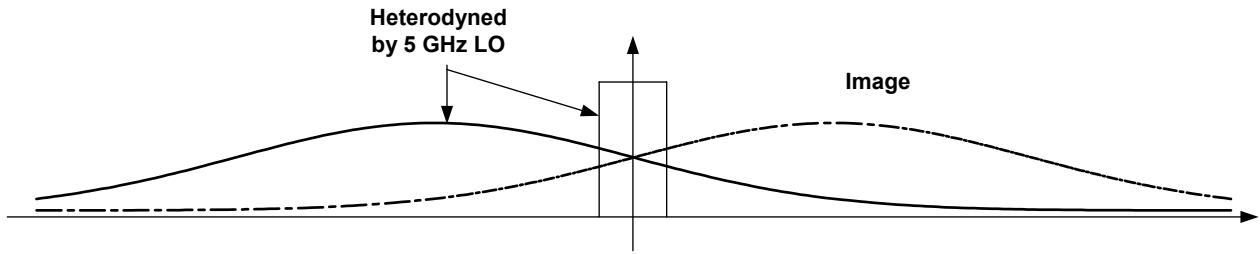


Figure 2.3.3 Overlap of downconverted UWB spectrum and its image.

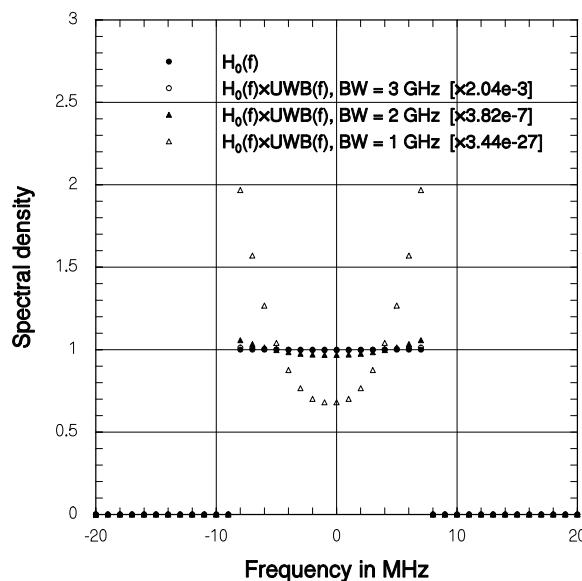


Figure 2.3.4 Example effective baseband UWB interference spectrum.

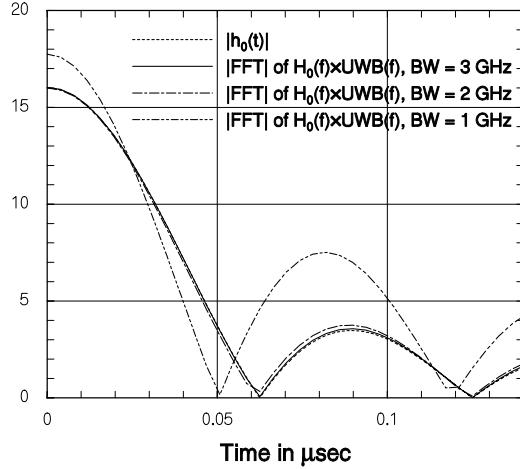


Figure 2.3.5 Example pulse shape of UWB baseband interference.

### 2.3.2 Effect at baseband of train of modulated UWB pulses

Prior to downconversion and filtering, in the absence of multipath reflections and ignoring propagation delay, the interfering UWB signal is modeled at the receiver as

$$z(t) = C_R \sum_{l=-\infty}^{\infty} u_l g(t - lT_u) \quad (2.3.10)$$

At baseband, in the quadratures the resulting interference is given by

$$z_I(t) = C_R \sum_{n=-\infty}^{\infty} u_l p_I(t - nT_u) \quad (2.3.11a)$$

and

$$z_Q(t) = C_R \sum_{l=-\infty}^{\infty} u_l p_Q(t - lT_u) \quad (2.3.11b)$$

The autocorrelation functions of these waveforms are given by [8]

$$R_{zI}(\tau) = \sum_{r=-\infty}^{\infty} R_u(r) R_{pI}(\tau - rT_u) = R_{pI}(\tau) * \sum_{r=-\infty}^{\infty} R_u(r) \delta(\tau - rT_u) \quad (2.3.12a)$$

and

$$R_{zQ}(\tau) = \sum_{r=-\infty}^{\infty} R_u(r) R_{pQ}(\tau - rT_u) = R_{pQ}(\tau) * \sum_{r=-\infty}^{\infty} R_u(r) \delta(\tau - rT_u) \quad (2.3.12b)$$

where  $R_u(r)$  is the discrete autocorrelation sequence of the data;  $R_{pI}(\tau)$  and  $R_{pQ}(\tau)$  are the (continuous) autocorrelation functions of the effective UWB pulses at baseband. If the data is completely random, then  $R_u(r)$  is zero except for  $r = 0$ , so that the autocorrelation functions and spectrums of the baseband UWB pulse trains are identical with the respective autocorrelation

functions and spectrums of the in-phase and quadrature pulses. At the other extreme, if the data is periodic, so that  $R_u(r)$  is periodic with period  $P$  symbols, then the power spectral density function of the effective UWB baseband interference involves the discrete Fourier transform of  $R_u(r)$  and the in-phase interference spectrum, for example, is given by

$$|Z_I(\omega)|^2 = |P_I(\omega)|^2 \frac{1}{(PT_u)^2} \sum_{k=-\infty}^{\infty} \text{DFT}_{R_u}\{k\} \cdot \delta\left(\omega - \frac{2\pi k}{PT_u}\right) \quad (2.3.13)$$

which is a spectrum of impulse functions (spectral lines) whose amplitudes are determined by the pulse spectrum and the DFT of the data autocorrelation sequence.

For a real UWB communication signal using periodic framing data, only a small portion of the data, if any, is repetitive framing, so that the signal has a continuous spectrum, possibly with some frequency peaks that resemble spectral lines [9]. Techniques such as dithering (varying the time between pulses pseudorandomly) and/or using pulse-position modulation can minimize the presence of lines in its spectrum and make the signal appear to be more “noiselike.” Methods exist for pseudorandomly encoding the framing data to remove such spectral lines.

### 2.3.3 Samples and FFT of the UWB interference

For an OFDM FFT size of 64, the sample rate equals  $64/3.2 \mu\text{s} = 20 \text{ Msps}$ , corresponding to  $\Delta_T = 50 \text{ ns}$ . Therefore I and Q samples of the UWB interference are given by

$$z_{mI} = z_I(m\Delta_T) = C_R \sum_{l=-\infty}^{\infty} u_l p_I(m\Delta_T - lT_u) \quad (2.3.14a)$$

and

$$z_{mQ} = z_Q(m\Delta_T) = C_R \sum_{l=-\infty}^{\infty} u_l p_Q(m\Delta_T - lT_u) \quad (2.3.14b)$$

Since the pulse duration,  $N_u T_r$ , can be on the order of one nanosecond or less, a pulse rate,  $1/T_u$ , on the order of 1 GHz is likely for the highest data rates. The implication is that many of the effective baseband pulses in (2.3.14a) and (2.3.14b) will overlap, perhaps 50 or more, to contribute to each sample. At the same time, because the baseband pulse energy is largely confined to the pulse interval, the significant overlap of the pulses prior to the beginning of the OFDM symbol and following the OFDM symbol will be due to UWB pulses arriving within one baseband filter period ( $1/B$ ) on either side of the OFDM symbol. Thus the relevant range of the index  $l$  in (2.3.14a) and (2.3.14b) is found to be

$$-L_1 \leq l \leq L_2 - 1 \quad \text{where } 0 \cdot \Delta_T - 1/B \approx -L_1 T_u \text{ and } 64 \cdot \Delta_T + 1/B \approx L_2 T_u \quad (2.3.15a)$$

or

$$L_1 \approx \lfloor 1/B T_u \rfloor \quad \text{and} \quad L_2 \approx L_1 + \lceil 64\Delta_T / T_u \rceil = L_1 + M_u \quad (2.3.15b)$$

where

$$M_u \triangleq \lceil 64\Delta_T / T_u \rceil = \lceil 1/\Delta_F T_u \rceil \quad (2.3.15c)$$

$M_u$  is the equivalent number of UWB pulses during the OFDM symbol interval, minus the guard time. For example, for  $B = 53\Delta_F = 53/64\Delta_T = 16.5625$  MHz and  $1/T_u = 250$  MHz, we have the values  $L_1 = \lfloor 250/16.5625 \rfloor = 15$ ,  $M_u = \lceil 3.2 \cdot 250 \rceil = 800$ , and  $L_2 = 815$ .

The contribution of the UWB interference to the  $i$ th OFDM output is found as the DFT of  $z_{mI} + jz_{mQ}$ , given by

$$\begin{aligned} U_i &= \frac{1}{N_p} \sum_{m=0}^{N_p-1} (z_{mI} + jz_{mQ}) e^{-j2\pi mi/N_p} \\ &= \frac{C_R}{N_p} \sum_{m=0}^{N_p-1} \sum_{l=-L_1}^{M_u+L_1-1} u_l [p_I(m\Delta_T - lT_u) + j p_Q(m\Delta_T - lT_u)] e^{-j2\pi mi/N_p} \\ &= \frac{C_R}{N_p} \sum_{l=-L_1}^{M_u+L_1-1} u_l \sum_{m=0}^{N_p-1} [p_I(m\Delta_T - lT_u) + j p_Q(m\Delta_T - lT_u)] e^{-j2\pi mi/N_p} \end{aligned} \quad (2.3.16a)$$

Substituting from (2.3.4) and (2.3.7), we have

$$\begin{aligned} U_i &= \frac{C_R}{N_p} \sum_{l=-L_1}^{M_u+L_1-1} u_l \sum_{m=0}^{N_p-1} e^{-j2\pi mi/N_p} \int_{-B/2}^{B/2} df e^{j2\pi f(m\Delta_T - lT_u)} e^{j\varphi} G(\omega_c) H_0(2\pi f) \\ &= C_R e^{j\varphi} G(\omega_c) \sum_{l=-L_1}^{M_u+L_1-1} u_l \int_{-B/2}^{B/2} df e^{-j2\pi f \cdot lT_u} H_0(2\pi f) \frac{1}{N_p} \sum_{m=0}^{N_p-1} e^{j2\pi m\Delta_T(f - i\Delta_F)} \\ &= C_R e^{j\varphi} G(\omega_c) H_0(2\pi i\Delta_F) \sum_{l=-L_1}^{M_u+L_1-1} u_l e^{-j2\pi i\Delta_F \cdot lT_u} \\ &= C_R e^{j\varphi} G(\omega_c) \sum_{l=-L_1}^{M_u+L_1-1} u_l e^{-j2\pi il/M_u} \end{aligned} \quad (2.3.16b)$$

The last factor in (2.3.16b) is practically the  $M_u$ -point DFT of the UWB signal data, so that  $U_i$  reflects the harmonic content of the pulse data during the OFDM symbol interval, at frequency  $f_i = i\Delta_F$  for positive values of the index. Note that at the OFDM frequencies below the center frequency, we have

$$\begin{aligned} U_{-i} &= C_R e^{j\varphi} G(\omega_c) H_0(-2\pi i\Delta_F) \sum_{l=-L_1}^{M_u+L_1-1} u_l e^{j2\pi i\Delta_F \cdot lT_u} \\ &= C_R e^{j\varphi} G(\omega_c) \sum_{l=-L_1}^{M_u+L_1-1} u_l e^{j2\pi il/M_u} \end{aligned} \quad (2.3.16c)$$

The expected value of the complex values of  $U_i$  equals zero because the UWB data are independent and equally likely to be plus or minus one. Its variance therefore is given by

$$\sigma_{Ui}^2 = \frac{1}{2} E\left\{ |U_i|^2 \right\} = \frac{1}{2} C_R^2 |G(\omega_c)|^2 E\left\{ \left| \sum_{l=-L_1}^{M_u+L_1-1} u_l e^{j2\pi il/M_u} \right|^2 \right\} = \frac{M_u + 2L_1}{2} C_R^2 |G(\omega_c)|^2 \quad (2.3.17)$$

This expression assumes that the UWB signal is on for the entire duration of the OFDM symbol.

### 3. PREDICTED PERFORMANCE

The OFDM receiver reverses the modulation, interleaving, and coding operations that were used to generate the signal. This process is subject to error because the signal is corrupted by receiver noise and, possibly by interference—in this case, a coexisting UWB signal. In this section we present predictions for the OFDM receiver performance in noise only and in the presence of interference, both with and without the error-control coding.

#### 3.1 OFDM Performance in Noise Only

##### 3.1.1 Probability of bit error without coding

When the OFDM carrier modulation is BPSK, as indicated in (A.1) the channel output per OFDM symbol is real-valued and is comprised of Gaussian noise with variance  $\sigma_{ni}^2 \doteq N_0 \Delta_F$ , according to (2.2.4), plus a signed signal value, either  $+\sqrt{2P_{Rc}}$  or  $-\sqrt{2P_{Rc}}$ , according to (2.1.4b). Assuming that the received signal value is negative, the probability of bit error in the Gaussian channel (with or without interleaving) is the probability that the noise value is large enough to change the polarity of the signal plus noise value to positive:

$$\begin{aligned} P_{e,BPSK} &= \Pr\left\{ \mathcal{N}_{nil} > \sqrt{2P_{Rc}} \mid b_{ni} = -1 \right\} \\ &= \Pr\left\{ \sigma_{ni}\gamma > \sqrt{2P_{Rc}} \right\} = \Pr\left\{ \gamma > \sqrt{\frac{2P_{Rc}}{\sigma_{ni}^2}} \right\} = Q\left( \sqrt{\frac{2P_{Rc}}{\sigma_{ni}^2}} \right) \end{aligned} \quad (3.1.1)$$

where  $\gamma$  represents a zero-mean, unit-variance Gaussian random variable and

$$Q(a) = \int_a^\infty \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx \quad (3.1.2)$$

is the complementary probability distribution function for the Gaussian distribution. We can write the SNR as

$$\frac{P_{Rc}}{\sigma_{ni}^2} = \frac{P_{Rc}}{N_0 \Delta_F} = \frac{(1/52)P_R}{N_0 (1/T_0)} = \frac{1}{52} \cdot \frac{P_R \cdot T_0}{N_0} = \frac{1}{52} \cdot \frac{P_R \cdot (4/5)T_s}{N_0} = \frac{4}{5} \cdot \frac{1}{52} \cdot \frac{P_R \cdot 48T_b}{N_0} = \frac{4}{5} \cdot \frac{48}{52} \cdot \frac{E_b}{N_0} \quad (3.1.3)$$

where, as denoted previously,  $T_0$  is the FFT time (symbol time minus the guard time),  $T_s$  is the OFDM symbol period, and  $T_b$  is the effective bit period. In general, for carrier modulations with alphabet size  $M = 2^K$  and coding with rate  $r$ , the SNR is equal to

$$\frac{P_{Rc}}{\sigma_{ni}^2} = F \cdot Kr \cdot \frac{E_b}{N_0}, \quad F = \frac{4}{5} \cdot \frac{48}{52} = 0.7385 \quad (3.1.4)$$

The factor  $F$  represents an OFDM implementation loss of 1.32 dB, due to the guard time and the use of pilot carriers.

When the OFDM carrier modulation is QPSK, for which  $K = 2$ , the channel output per OFDM symbol is complex. As indicated in (A.2), the real and imaginary parts (quadratures) of the channel output are independent bits with independent noise components. The signal component of each quadrature, according to the normalization by the factor  $\kappa$  given in (1.1.1b), is  $\pm\sqrt{2P_{Rc}}/\kappa = \pm\sqrt{P_{Rc}}$ , so the probability of bit error is given by

$$P_{e,QPSK} = Q\left(\sqrt{\frac{P_{Rc}}{\sigma_{ni}^2}}\right) = Q\left(\sqrt{F \cdot Kr \cdot \frac{E_b}{N_0}}\right) = Q\left(\sqrt{F \cdot 2 \cdot \frac{E_b}{N_0}}\right) \quad (3.1.5)$$

Thus the probability of bit error is the same for QPSK as it is for BPSK. Again, since each demodulated symbol has independent noise, the probability of bit error for OFDM/QPSK in the Gaussian channel is the same with and without interleaving.

When the OFDM carrier modulation is 16-QAM or 64-QAM, for which  $K = 4$  or  $K = 6$ , respectively, the channel output per OFDM symbol is complex. As indicated in (A.3), for 16-QAM the real and imaginary parts (quadratures) of the channel output are functions of independent pairs of bits. Each pair of bits determines the symbol values of 4-ary pulse amplitude modulation (PAM) symbols with independent noise components per quadrature. From Figure A.1, we observe for both quadratures that the first of the two bits is determined by the polarity of the received symbol, and the second bit is determined by the magnitude of the symbol. The signal component of each quadrature, according to the normalization by the factor  $\kappa$  given in (1.1.1b), takes the values  $\pm\sqrt{2P_{Rc}}/\kappa = \pm\sqrt{P_{Rc}}/5$  and  $\pm 3\sqrt{2P_{Rc}}/\kappa = \pm 3\sqrt{P_{Rc}}/5$ , so the probability of bit error is given by the average of the following conditional bit error probabilities:

$$\begin{aligned} \Pr\{\text{bit 1 error} | 00\} &= \Pr\{\text{bit 1 error} | 10\} = Q\left(3\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \\ \Pr\{\text{bit 1 error} | 01\} &= \Pr\{\text{bit 1 error} | 11\} = Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \\ \Pr\{\text{bit 2 error} | 00\} &= \Pr\{\text{bit 2 error} | 10\} = Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) - Q\left(5\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \\ \Pr\{\text{bit 2 error} | 01\} &= \Pr\{\text{bit 2 error} | 11\} = Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) + Q\left(3\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \end{aligned} \quad (3.1.6)$$

Note that the bit 1 and bit 2 error events are not independent. The average bit error probability is

$$P_{e,16\text{-QAM}} = \frac{1}{4} \left\{ 3Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) + 2Q\left(3\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) - Q\left(5\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \right\} \doteq \frac{3}{4}Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \quad (3.1.7a)$$

In view of (3.1.4), this result can be expressed as

$$P_{e,16\text{-QAM}} \doteq \frac{3}{4}Q\left(\sqrt{F \cdot \frac{4}{5} \frac{rE_b}{N_0}}\right) = \frac{3}{4}Q\left(\sqrt{F \cdot \frac{4}{5} \frac{E_b}{N_0}}\right) \quad (3.1.7b)$$

This result agrees with the approach in [4], based on finding the symbol error probability in each quadrature, then the overall symbol error probability, and relating the bit error probability to the symbol error probability. According to this procedure, each 4-ary quadrature has the conditional symbol error probabilities

$$\begin{aligned} \Pr\{\text{symbol error} | 00\} &= \Pr\{\text{symbol error} | 10\} = Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \\ \Pr\{\text{symbol error} | 01\} &= \Pr\{\text{symbol error} | 11\} = 2Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \end{aligned} \quad (3.1.8a)$$

leading to the average symbol error probability per quadrature of

$$\Pr\{\text{quadrature symbol error}\} = P_{qs} = \frac{3}{2}Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \quad (3.1.8b)$$

and the total symbol error of

$$\begin{aligned} \Pr\{\text{symbol error}\} &= P_s = 1 - (1 - P_{qs})^2 \\ &= 3Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) - \frac{9}{4}Q^2\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \doteq 3Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) \end{aligned} \quad (3.1.8c)$$

and the bit error is converted from the symbol error probability to obtain

$$\Pr\{\text{bit error}\} = \frac{1}{K} \Pr\{\text{symbol error}\} \doteq \frac{3}{4}Q\left(\sqrt{\frac{P_{Rc}}{5\sigma_{ni}^2}}\right) = \frac{3}{4}Q\left(\sqrt{F \cdot \frac{4}{5} \frac{E_b}{N_0}}\right) \quad (3.1.8d)$$

This expression for the bit error probability of course agrees with (3.1.7a).

As indicated in (A.4), for 64-QAM the quadratures of the channel output are determined by independent triples of bits. Each triple determines the symbol values of 8-ary PAM symbols

with independent noise components per quadrature. From Figure A.1, we observe for both quadratures that the first of the three bits is determined by the polarity of the received symbol, and the second bit is determined by the magnitude of the symbol; the third bit is determined by a combination of magnitude and polarity. The signal component of each quadrature, according to the normalization by the factor  $\kappa$ , takes the values

$$\begin{aligned}\pm\sqrt{2P_{Rc}}/\kappa &= \pm\sqrt{P_{Rc}/41}, & \pm3\sqrt{2P_{Rc}}/\kappa &= \pm3\sqrt{P_{Rc}/21}, \\ \pm5\sqrt{2P_{Rc}}/\kappa &= \pm5\sqrt{P_{Rc}/21}, & \pm7\sqrt{2P_{Rc}}/\kappa &= \pm7\sqrt{P_{Rc}/21},\end{aligned}$$

so the probability of bit error is given by the average of the following conditional bit error probabilities, using  $X = \sqrt{P_{Rc}/21\sigma_{ni}^2}$ :

$$\begin{aligned}\Pr\{\text{bit 1 error} | 000\} &= Q(7X), & \Pr\{\text{bit 1 error} | 001\} &= Q(5X), \\ \Pr\{\text{bit 1 error} | 011\} &= Q(3X), & \Pr\{\text{bit 1 error} | 010\} &= Q(X)\end{aligned}\tag{3.1.9a}$$

$$\begin{aligned}\Pr\{\text{bit 2 error} | 000\} &= Q(3X) - Q(11X), & \Pr\{\text{bit 2 error} | 001\} &= Q(X) - Q(9X) \\ \Pr\{\text{bit 2 error} | 011\} &= Q(X) + Q(7X), & \Pr\{\text{bit 2 error} | 010\} &= Q(3X) + Q(5X)\end{aligned}\tag{3.1.9b}$$

$$\begin{aligned}\Pr\{\text{bit 3 error} | 000\} &= Q(X) - Q(5X) + Q(9X) - Q(13X) \\ \Pr\{\text{bit 3 error} | 001\} &= Q(X) + Q(3X) - Q(7X) + Q(11X) \\ \Pr\{\text{bit 3 error} | 011\} &= Q(3X) + Q(X) - Q(5X) + Q(9X) \\ \Pr\{\text{bit 3 error} | 010\} &= Q(X) - Q(5X) + Q(3X) - Q(7X)\end{aligned}\tag{3.1.9c}$$

Note that the bit errors are not independent. These expressions lead to the following average bit error probability for 64-QAM:

$$\begin{aligned}P_{e,64-\text{QAM}} &= \frac{1}{12}\{7Q(X) + 6Q(3X) - Q(5X) + Q(9X) - Q(13X)\} \\ &\doteq \frac{7}{12}Q\left(\sqrt{\frac{P_{Rc}}{21\sigma_{ni}^2}}\right) = \frac{7}{12}Q\left(\sqrt{F \cdot \frac{6}{21} \frac{rE_b}{N_0}}\right) = \frac{7}{12}Q\left(\sqrt{F \cdot \frac{6}{21} \frac{E_b}{N_0}}\right)\end{aligned}\tag{3.1.10}$$

Figure 3.1.1 shows the uncoded bit error probabilities for the four OFDM modulations. Note that the differences in the BER performance among the modulations cannot be expressed exactly as a constant value of dB of  $E_b/N_0$ , but from Figure 3.1.1 it is clear that 16-QAM is about 3.8 dB worse than BPSK or QPSK and 64-QAM is about 8.1 dB worse.

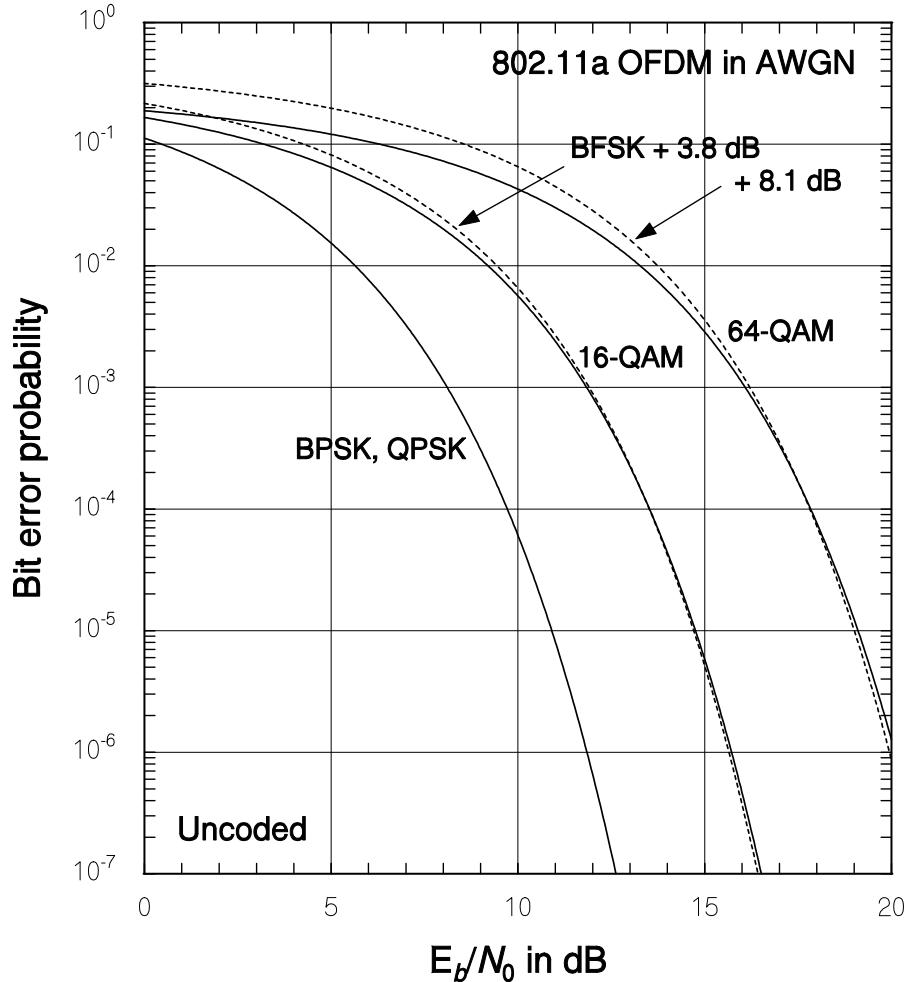


Figure 3.1.1 Uncoded BERs for 802.11a OFDM modulations in noise only.

### 3.1.2 Probability of bit error with coding

The coding used in IEEE 802.11a is rate-1/2, constraint length 7 convolutional coding with optional puncturing to achieve code rates of 2/3 and 3/4. Interleaving is used to protect against burst errors and, in the case of QAM-16 and QAM-64 OFDM carrier modulation and soft decoding, to randomize the dependencies among the errors in demodulator code symbol outputs.

#### 3.1.2.1 BPSK and QPSK

An upper bound for the unpunctured bit error probability using the code with BPSK or QPSK and soft decisions is given by [4]

$$P_{e,BPSK} < \sum_{d=10}^{\infty} \beta_d Q\left(\sqrt{d \cdot \frac{E_b}{N_0}}\right) \quad (3.1.11)$$

where the weights  $\{\beta_d\}$ , from [10], are shown in Table 3.1.1. For hard-decision decoding, a Chernoff upper bound for the BER is given by [11, 12]

$$P_{e,\text{BPSK}} < \frac{1}{2} \sum_{d=10}^{\infty} \beta_d D^d, \quad D = \sqrt{4p(1-p)} \quad (3.1.12a)$$

where  $p$  is the uncoded error rate (3.1.5) with  $r = 1/2$ . A tighter

To achieve code rates of  $2/3$  and  $3/4$ , the output of the rate  $1/2$  convolutional coder is modified by, respectively, deleting one out each four coded bits or two out of each six coded bits. The performance of the code for BPSK or QPSK with soft decisions is given by the following upper bound, using the distances and weights given in Table 3.1.2, extracted from [13]:

$$P_{e,\text{BPSK}} < \frac{1}{b} \sum_{d=d_{\min}}^{\infty} \beta_d Q\left(\sqrt{d \cdot \frac{2rE_b}{N_0}}\right), \quad b = \begin{cases} 2, & r = 2/3 \\ 3, & r = 3/4 \end{cases} \quad (3.1.13)$$

For hard-decision decoding of the punctured codes, we use the Chernoff bound to write

$$P_{e,\text{BPSK}} < \frac{1}{2b} \sum_{d=d_{\min}}^{\infty} \beta_d D^d, \quad D = \sqrt{4p(1-p)} \quad (3.1.14)$$

where  $p$  is the uncoded error rate (3.1.5) with  $r = 2/3$  or  $3/4$ . These upper bounds on the bit error probability are shown in Figure 3.1.2 for ordinary BPSK or QPSK modulation, and in Figure 3.1.3 for the parameters of the BPSK and QPSK modulation in 802.11a OFDM, which does not use the  $2/3$  rate.

### 3.1.2.2 QAM

When the per-channel OFDM modulation is 16-QAM or 64-QAM, the receiver implements a 4-PAM or 8-PAM demodulator in each quadrature, respectively. For hard-decision decoding, the coded bits coming out of the demodulator all have the same weight, so the appropriate upper bounds are

$$P_{e,\text{QAM}} < \frac{1}{2b} \sum_{d=d_{\min}}^{\infty} \beta_d D^d, \quad D = \sqrt{4p(1-p)} \quad (3.1.15)$$

Table 3.1.1 Nonzero weights for rate  $1/2$  convolutional code used in 802.11a

$d$	Weight, $\beta_d$	$d$	Weight, $\beta_d$
10	36	20	502,690
12	211	22	3,322,763
14	1,404	24	21,292,910
16	11,633	26	134,365,911
18	77,433		

Table 3.1.2 Nonzero weights for rate 2/3 and rate 3/4 convolutional codes used in 802.11a

d	Weight, $\beta_d$		d	Weight, $\beta_d$	
	$r = 2/3$	$r = 3/4$		$r = 2/3$	$r = 3/4$
5		42	11	27,128	2,253,373
6	3	201	12	117,019	13,073,811
7	70	1,492	13	498,860	75,152,755
8	285	10,469	14	2,103,891	428,005,675
9	1,276	62,935	15	8,784,123	
10	6,160	379,644			

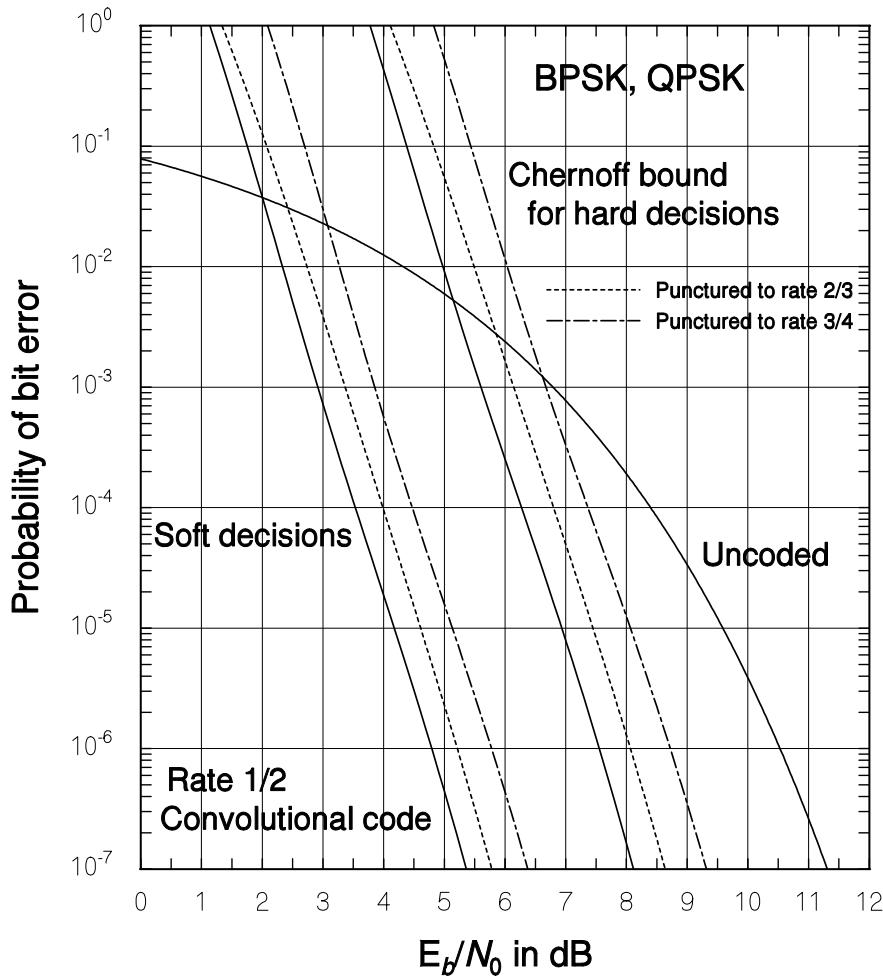


Figure 3.1.2 BER performance of BPSK and QPSK with rate – 1/2 convolutional coding

where  $p$  is the uncoded error rate (3.1.7b) for 16-QAM or (3.1.10) for 64-QAM, with  $r = 1/2, 2/3$  or  $3/4$ , and  $b = 1$  for  $r = 1/2$ .

For soft-decision decoding of OFDM/16-QAM, the coded 4-PAM bits coming out of the demodulator are actually weighted values (metrics) whose distributions vary with the position of the coded bits in the 4-PAM symbol [14]. The normalized value of the metrics for a quadrature

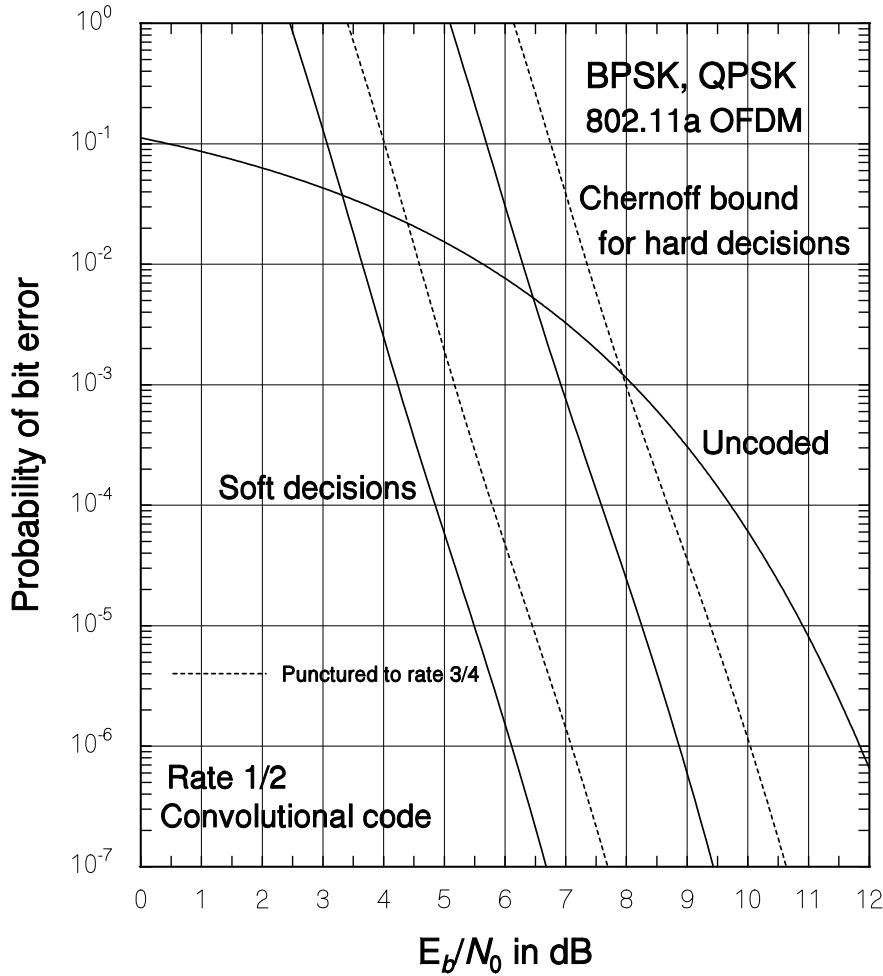


Figure 3.1.3 BER performance of 802.11a OFDM using BPSK and QPSK with rate – 1/2 convolutional coding

4-PAM symbol when 16-QAM is used is shown in Figure 3.1.4; the Euclidean distance (distance to sign-changing threshold) of the normalized metric for bit 0 is equally likely to be either 1 or 3, and for bit 1 it is always 1. Overlooking the discontinuities in the metric transformations in Figure 3.1.4, the average probability density function for a metric value is approximately a Gaussian mixture:

$$p_m(x) \approx \frac{3}{4} \cdot p_G\left(\frac{x - \sqrt{2P_{Rc}}/\kappa}{\sigma_{ni}}\right) + \frac{1}{4} \cdot p_G\left(\frac{x - 3\sqrt{2P_{Rc}}/\kappa}{\sigma_{ni}}\right) \quad (3.1.16a)$$

for which the characteristic function is

$$\varphi_m(v) = \frac{3}{4} \cdot e^{jv\sqrt{2P_{Rc}}/\kappa - v^2\sigma_{ni}^2/2} + \frac{1}{4} \cdot e^{j3v\sqrt{2P_{Rc}}/\kappa - v^2\sigma_{ni}^2/2} \quad (3.1.16b)$$

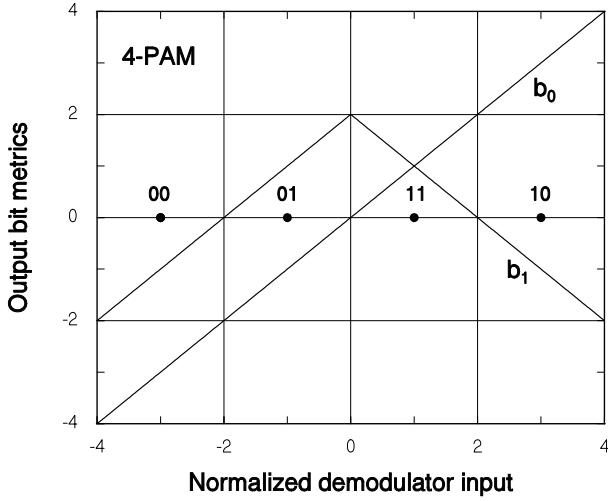


Figure 3.1.4 Quadrature bit outputs of the demodulator for 16-QAM

The characteristic function for the distribution of the sum ( $y_d$ ) of  $d$  soft-decision metrics for the 4-PAM symbols is given by the expression in (3.1.16b) raised to the power  $d$ , which can be reduced to the expression

$$\varphi_{yd}(\nu) = [\varphi_m(\nu)]^d = \left(\frac{3}{4}\right)^d e^{jdv\sqrt{2P_{Rc}} - d\nu^2\sigma_{ni}^2/2} \left(1 + \frac{1}{3} \cdot e^{j2\nu\sqrt{2P_{Rc}}}\right)^d \quad (3.1.16c)$$

so that the probability of error for a  $d$ -bit path (coded bits) is approximately

$$\Pr\{y_d < 0\} = \left(\frac{3}{4}\right)^d Q\left(\frac{\sqrt{d \cdot 2P_{Rc}}}{\kappa\sigma_{ni}}\right) = \left(\frac{3}{4}\right)^d Q\left(\sqrt{d \frac{2Kr E_b}{\kappa^2 N_0}}\right) \quad (3.1.16d)$$

The corresponding upper bound for soft-decision decoding of the 16-QAM OFDM modulation is given by

$$P_{e,16\text{-QAM}} < \frac{1}{b} \sum_{d=d_{\min}}^{\infty} \beta_d \left(\frac{3}{4}\right)^d Q\left(\sqrt{d \cdot \frac{4}{5} \cdot \frac{rE_b}{N_0}}\right) \quad (3.1.17)$$

Uncoded and coded BER bounds are shown in Figure 3.1.6 and Figure 3.1.7, respectively, for ordinary 16-QAM and for the parameters of the 16-QAM in 802.11a OFDM. Rate 2/3 coding is omitted from Figure 3.1.7 because it is not used in 802.11a.

Similarly, for soft-decision decoding of OFDM/64-QAM, the coded 16-PAM bits coming out of the demodulator are weighted values (metrics) whose distributions vary with the position of the coded bits in the 16-PAM symbol. The normalized value of the metrics for a quadrature 16-PAM symbol when 64-QAM is used is shown in Figure 3.1.5; the Euclidean distance (distance

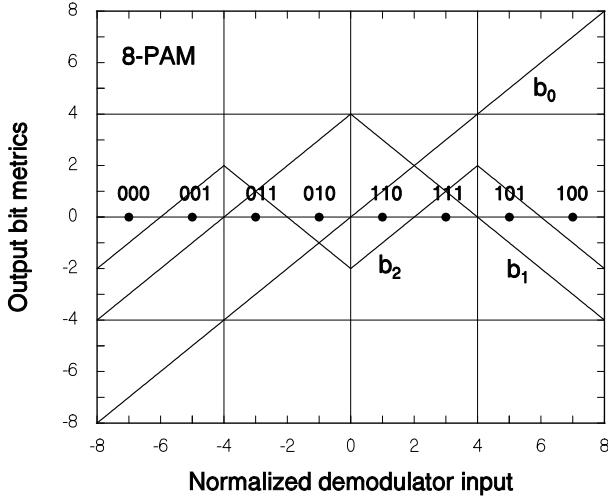


Figure 3.1.5 Quadrature bit outputs of the demodulator for 64-QAM

to sign-changing threshold) of the normalized metric for bit 0 is equally likely to be 1 or 3 or 5 or 7, for bit 1 is equally likely to be 1 or 3, and for bit 2 it is always 1. Overlooking the discontinuities in the metric transformations in Figure 3.1.5, the average probability density function for a metric value is approximately a Gaussian mixture:

$$p_m(x) \approx \frac{7}{12} \cdot p_G\left(\frac{x - \sqrt{2P_{Rc}}/\kappa}{\sigma_{ni}}\right) + \frac{3}{12} \cdot p_G\left(\frac{x - 3\sqrt{2P_{Rc}}/\kappa}{\sigma_{ni}}\right) \\ + \frac{1}{12} \cdot p_G\left(\frac{x - 5\sqrt{2P_{Rc}}/\kappa}{\sigma_{ni}}\right) + \frac{1}{12} \cdot p_G\left(\frac{x - 7\sqrt{2P_{Rc}}/\kappa}{\sigma_{ni}}\right) \quad (3.1.18a)$$

for which the characteristic function is

$$\varphi_m(v) = \frac{7}{12} \cdot e^{jv\sqrt{2P_{Rc}}/\kappa - v^2\sigma_{ni}^2/2} + \frac{3}{12} \cdot e^{j3v\sqrt{2P_{Rc}}/\kappa - v^2\sigma_{ni}^2/2} \\ + \frac{1}{12} \cdot e^{j5v\sqrt{2P_{Rc}}/\kappa - v^2\sigma_{ni}^2/2} + \frac{1}{12} \cdot e^{j7v\sqrt{2P_{Rc}}/\kappa - v^2\sigma_{ni}^2/2} \quad (3.1.18b)$$

The characteristic function for the distribution of the sum ( $y_d$ ) of  $d$  soft-decision metrics for the 64-PAM symbols is given by the expression in (3.1.18b) raised to the power  $d$ , which can be reduced to the expression

$$\varphi_{yd}(v) = [\varphi_m(v)]^d = \left(\frac{7}{12}\right)^d e^{jd\sqrt{2P_{Rc}} - d\sigma_{ni}^2/2} \left(1 + \frac{3}{7} \cdot e^{j2\sqrt{2P_{Rc}}} + \frac{1}{7} \cdot e^{j4\sqrt{2P_{Rc}}} + \frac{1}{7} \cdot e^{j6\sqrt{2P_{Rc}}}\right)^d \quad (3.1.18c)$$

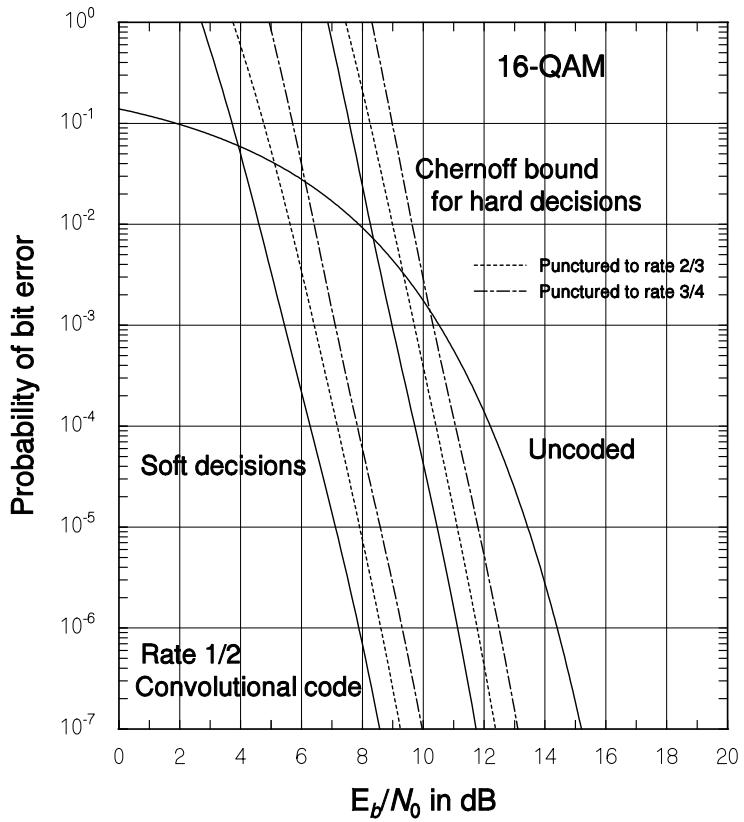


Figure 3.1.6 Uncoded and coded BER for 16-QAM.

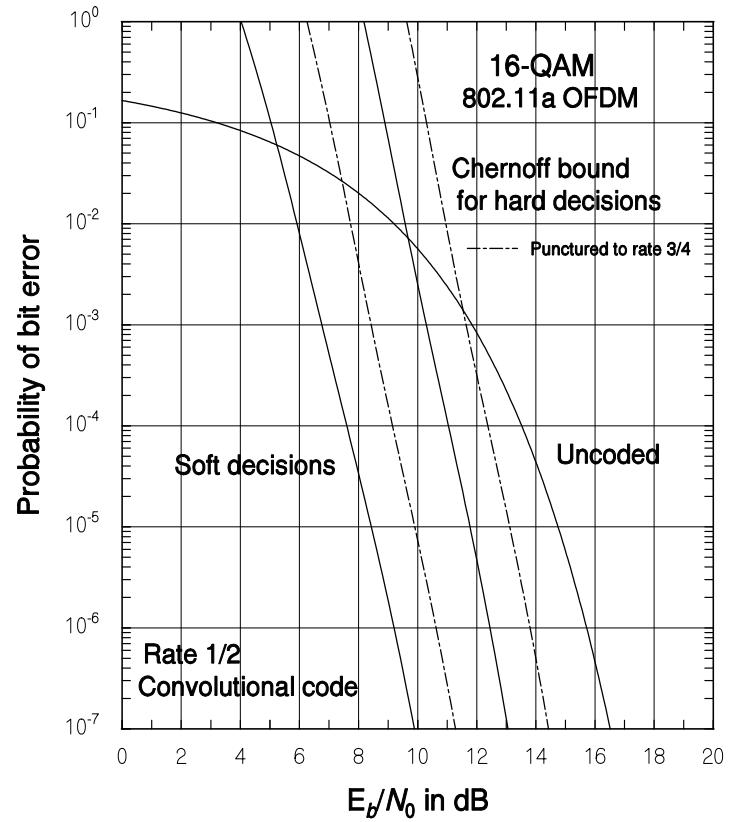


Figure 3.1.7 Uncoded and coded BER performance of 802.11a OFDM using 16-QAM.

so that the probability of error for a  $d$ -bit path (coded bits) is approximately

$$\Pr\{y_d < 0\} = \left(\frac{7}{12}\right)^d Q\left(\frac{\sqrt{d \cdot 2P_{rc}}}{\kappa \sigma_{ni}}\right) = \left(\frac{7}{12}\right)^d Q\left(\sqrt{d \frac{2Kr}{\kappa^2} \frac{E_b}{N_0}}\right) \quad (3.1.18d)$$

The corresponding upper bound for soft-decision decoding of the 16-QAM OFDM modulation is

$$P_{e,64\text{-QAM}} < \frac{1}{b} \sum_{d=d_{\min}}^{\infty} \beta_d \left(\frac{7}{12}\right)^d Q\left(\sqrt{d \cdot \frac{2}{7} \cdot \frac{rE_b}{N_0}}\right) \quad (3.1.19)$$

Uncoded and coded BER bounds are shown in Figure 3.1.8 for ordinary 64-QAM and in Figure 3.1.9 for the parameters of the 64-QAM in 802.11a OFDM. Rate 1/2 coding is omitted from 3.19 because it is not used in 802.11a. The bound for rate 1/2 coding in Figure 3.1.9 is evidently not valid for lower values of  $E_b/N_0$ , but asymptotically it appears to be realistic.

## 3.2 Performance of OFDM in Noise and UWB Interference

### 3.2.1 UWB without multipath

According to (2.2.4), the receiver noise power (variance) in each OFDM symbol FFT output bin equals  $\sigma_{ni}^2 \approx N_0 \Delta_F$  and according to (2.3.17), the variance of the UWB interference in an FFT output bin equals

$$\sigma_{Ui}^2 = \frac{M_u + 2L_1}{2} C_R^2 |G(\omega_c)|^2 \quad (3.2.1)$$

where

$$M_u + 2L_1 \approx \frac{1}{\Delta_F T_u} + \frac{2}{BT_u} = \frac{1}{\Delta_F T_u} + \frac{2}{53\Delta_F T_u} = \frac{1.04}{\Delta_F T_u} = 1.04M_u \quad (3.2.2)$$

Therefore, when the UWB interference is present, it increases the effective noise power by the ratio

$$\frac{\sigma_{ni}^2 + \sigma_{Ui}^2}{\sigma_{ni}^2} = 1 + \frac{\sigma_{Ui}^2}{\sigma_{ni}^2} = 1 + 1.04M_u \frac{C_R^2 |G(\omega_c)|^2}{2N_0 \Delta_F}, \text{ no multipath} \quad (3.2.3)$$

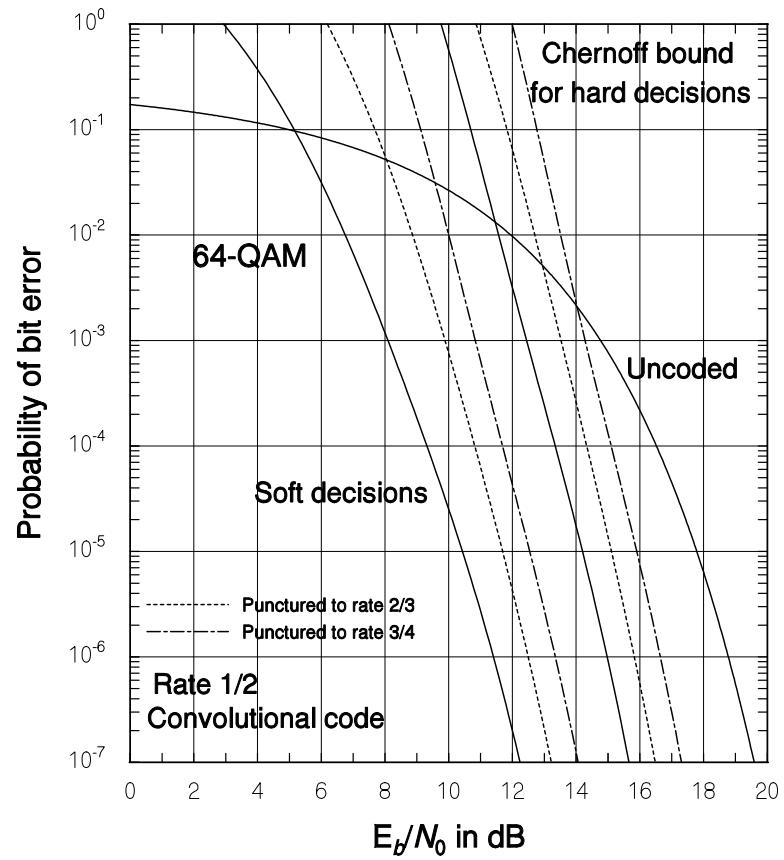


Figure 3.1.8 Uncoded and coded BER performance of 64-QAM.

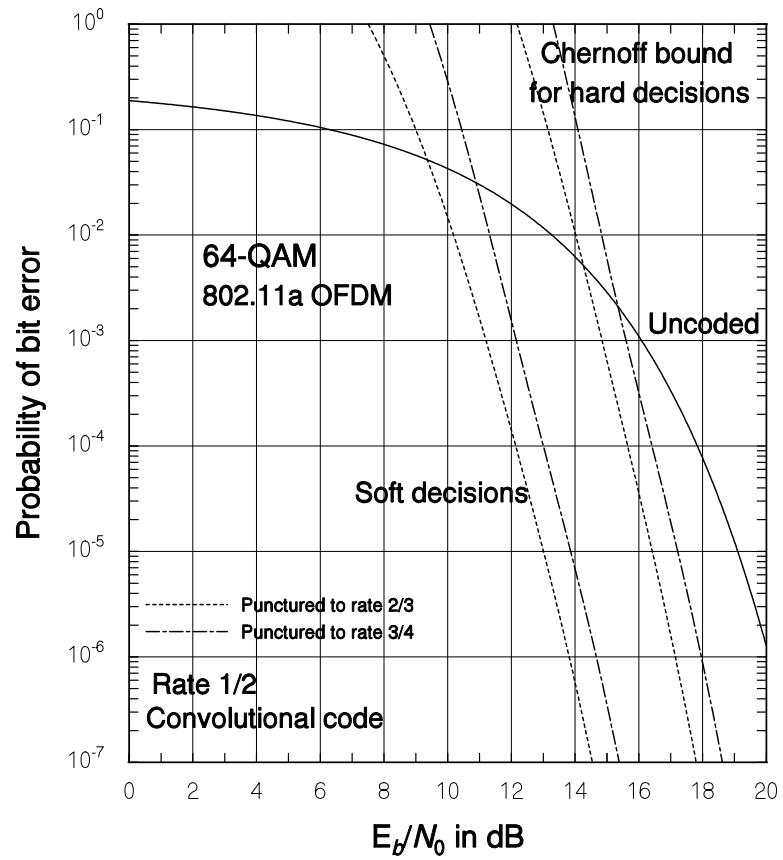


Figure 3.1.9 Uncoded and coded BER performance of 802.11a OFDM using 64-QAM.

### 3.2.2 UWB with multipath

Measurements have shown that many replicas of each transmitted UWB pulse can arrive at the receiver location over different reflective paths [16]. The result of this multipath activity is that the amount of UWB energy received during an OFDM symbol period is increased by some factor that is a function of the UWB transmitter location and 802.11a receiver location as well as other properties of the local environment. The additional energy is received in the form of resolvable (time-distinct) pulses that the UWB receiver may or may not be able to exploit by collecting and aligning them in time for optimal signal processing of the multipath signal. To the 802.11a receiver, of course, all the additional UWB energy is additional noise.

Pending further study on how to characterize the degree of multipath in various scenarios, we define the factor  $\mu \geq 1$  as the ratio of the received UWB power with multipath to the received UWB power without multipath, and re-write (3.2.3) as

$$\frac{\sigma_{ni}^2 + \mu\sigma_{Ui}^2}{\sigma_{ni}^2} = 1 + \mu \frac{\sigma_{Ui}^2}{\sigma_{ni}^2} = 1 + 1.04\mu M_u \frac{C_R^2 |G(\omega_c)|^2}{2N_0\Delta_F}, \text{ multipath} \quad (3.2.4)$$

## 4. VALIDATION OF SIMULATION

Using the predicted OFDM performance, we now provide validation results for the IEEE 802.11a OFDM simulation described in [17].

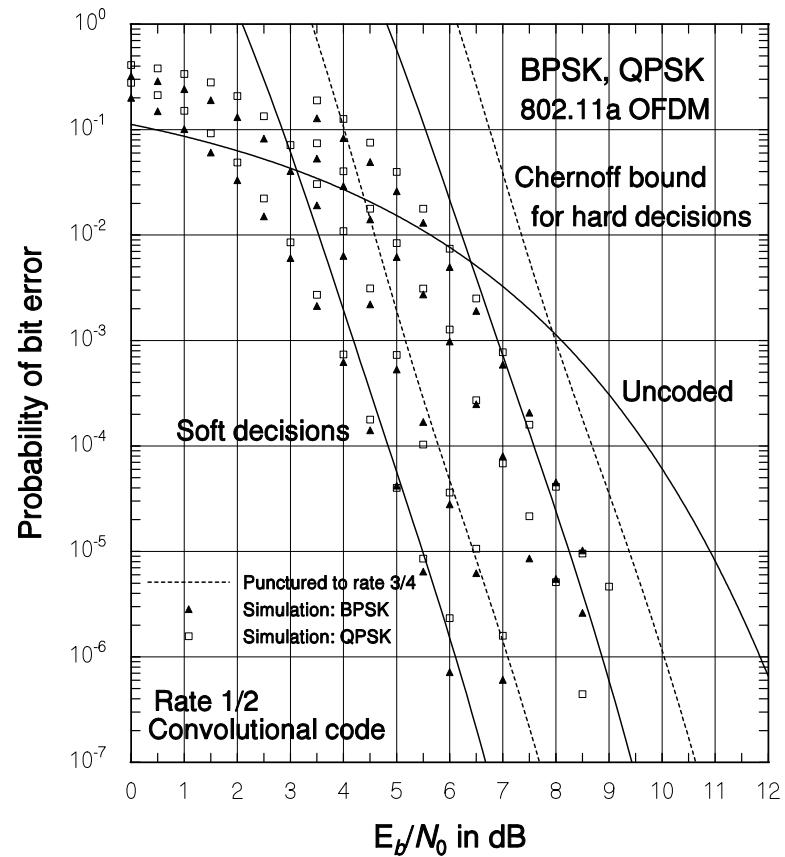
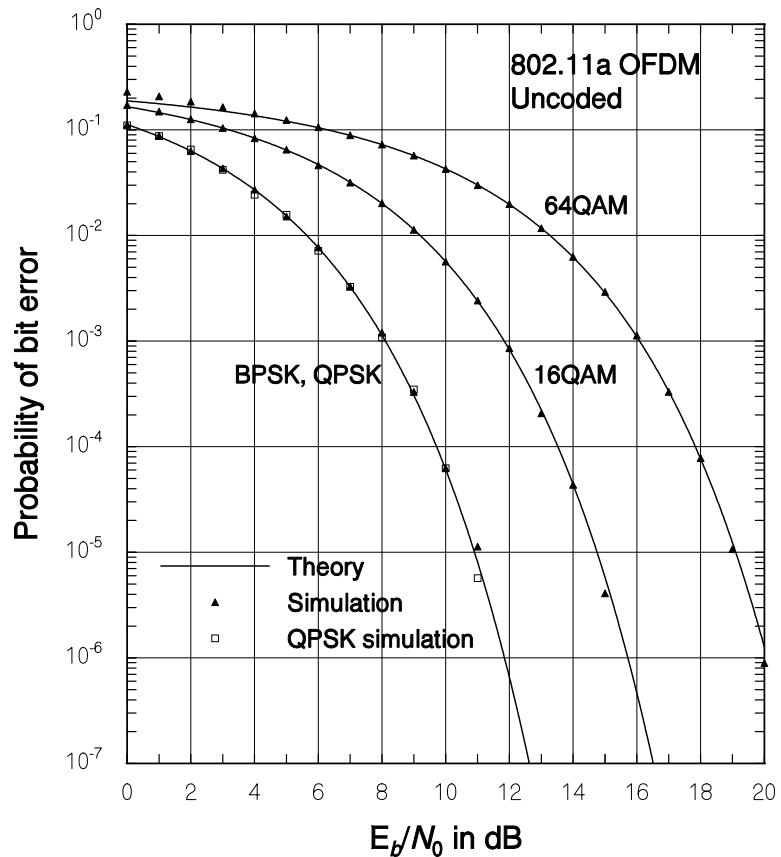
### 4.1 Performance on the AWGN Channel

The simulation program implements AWGN channel conditions by adding independent noise samples to the quadratures of the OFDM transmitter output. The FFT is not normalized by  $1/N_p$  but the IFFT is. For simplicity the received OFDM signal amplitude is taken to be the same as the transmitted amplitude, with each channel having an average output power of  $P_{Rc} = 1/2$ . Based on (3.1.4), a target value of  $E_b/N_0$  is simulated by making the sample noise variance

$$\sigma_v^2 = \frac{\sigma_{ni}^2}{N_p} = \left( \frac{E_b}{N_0} \right)_{\text{target}}^{-1} \frac{P_{Rc}}{KrF} \cdot \frac{1}{N_p} = \left( \frac{E_b}{N_0} \right)_{\text{target}}^{-1} \frac{1}{2Kr} \cdot \frac{5 \cdot 52}{4 \cdot 48} \cdot \frac{1}{N_p} = \left( \frac{E_b}{N_0} \right)_{\text{target}}^{-1} \frac{65}{96Kr} \cdot \frac{1}{N_p} \quad (4.1)$$

As shown in Figure 4.1.1, the uncoded performance determined by the simulation is very close to the theoretical value. The number of OFDM symbols and bits used in the error probability calculation were as shown in Table C.1 in the appendix, based on simulating two OFDM symbols per packet. Sixteen bits of each packet were excised in the simulation (2 at the beginning and 14 at the end).

Tables C.2 and C.3 show the numbers of packets used to simulate coded OFDM symbols on the AWGN channel, with the error probability results shown in Figure 4.1.2 for BPSK and QPSK, Figure 4.1.3 for 16-QAM, and in Figure 4.1.4 for 64-QAM. The number of bits per packet equals  $96Kr$ , less six tail bits per packet.



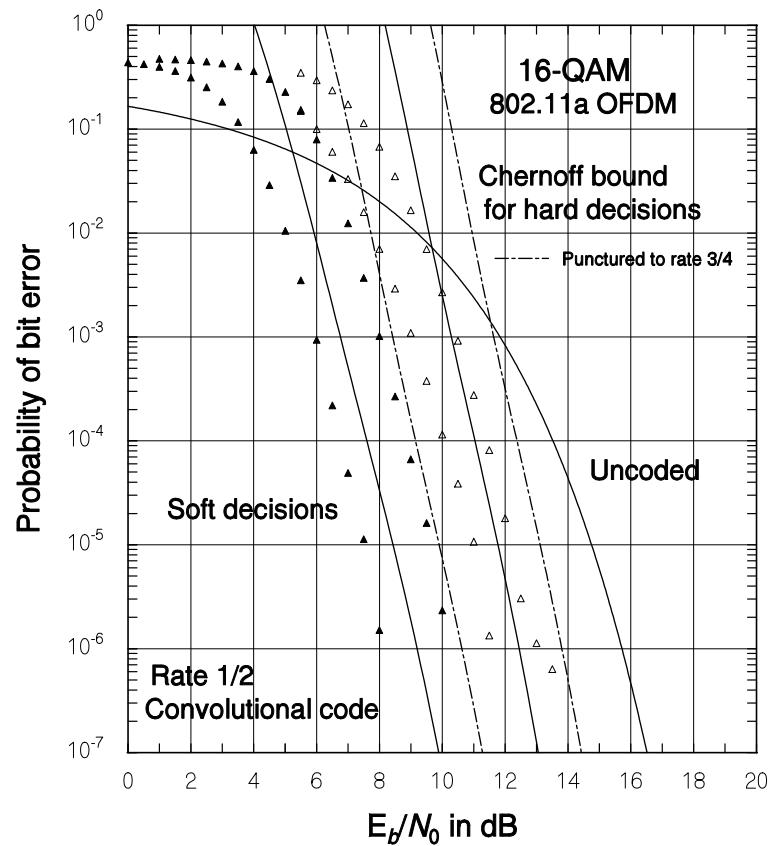


Figure 4.1.3 Comparison of IEEE 802.11a theoretical and simulated 16-QAM BERs with coding on the AWGN channel.

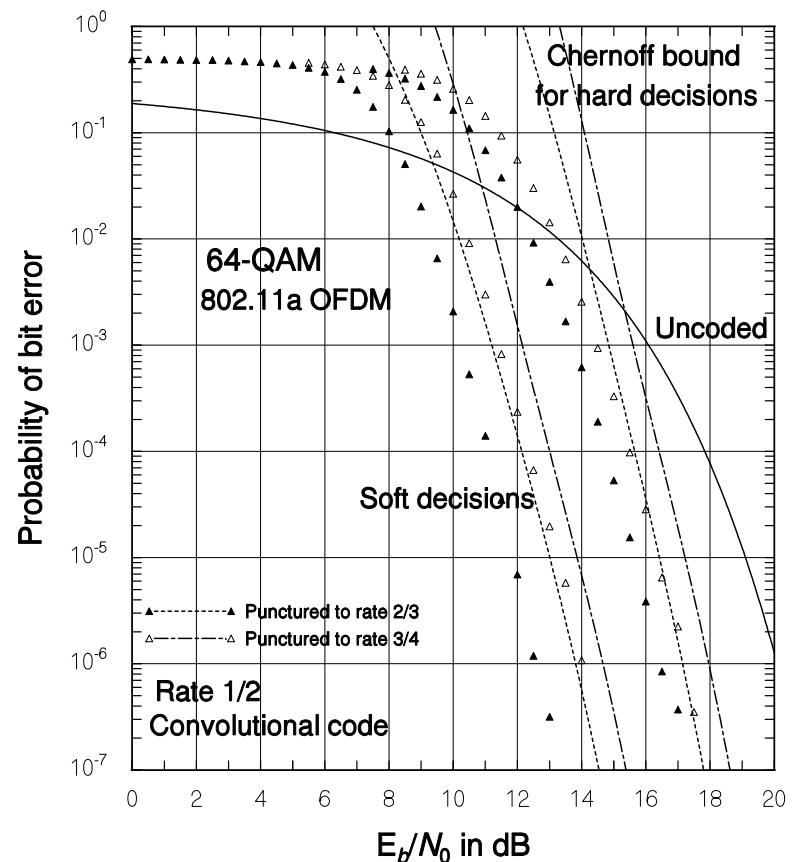


Figure 4.1.4 Comparison of IEEE 802.11a theoretical and simulated 64-QAM BERs with coding on the AWGN channel.

## 4.2 Performance with UWB Interference

The simulation program implements UWB interference conditions by generating independent pulse modulation samples to form the UWB signal given by:

$$z_{ml} = z_I(m\Delta_T) = C_R \sum_{l=-\infty}^{\infty} u_l p_I(m\Delta_T - lT_u) = C_{R1} \cos \varphi \sum_{l=-\infty}^{\infty} u_l h_0(m\Delta_T - lT_u) \quad (4.2a)$$

and

$$z_{mQ} = z_Q(m\Delta_T) = C_R \sum_{l=-\infty}^{\infty} u_l p_Q(m\Delta_T - lT_u) = C_{R1} \sin \varphi \sum_{l=-\infty}^{\infty} u_l h_0(m\Delta_T - lT_u) \quad (4.2b)$$

where  $C_{R1}$  is the effective amplitude of the interfering pulses and  $h_0(t)$  is the impulse response of the receiver baseband filters. The effective signal energy-to-interference density ratio at the receiver's FFT output is

$$\frac{E_b}{N_I} = \frac{P_R / R_b}{\sigma_{Ui}^2 / \Delta_F} = \frac{P_R}{\sigma_{Ui}^2} \cdot \frac{\Delta_F}{R_b} = \frac{52P_{Rc}}{\sigma_{Ui}^2} \cdot \frac{1/3.2 \mu s}{12Kr \text{MHz}} = \frac{65}{48Kr} \cdot \frac{P_{Rc}}{\sigma_{Ui}^2} \quad (4.3)$$

Solving this equation for the received UWB signal power in an FFT output bin and relating it to the UWB power at the receiver input yields the following expression involving the effective pulse amplitude, using the reference value of  $P_{Rc} = 1/2$ :

$$\sigma_{Ui}^2 = \left( \frac{E_b}{N_I} \right)_{\text{target}}^{-1} \frac{65}{96Kr} \approx N_p \cdot \frac{1}{2} M_u C_{R1}^2 = N_p \cdot \frac{\Delta_T}{2T_u} C_{R1}^2 = \frac{T_s}{2T_u} C_{R1}^2 \quad (4.4a)$$

where  $T_s$  denotes the duration of the OFDM symbol (not including guard time). This equation ignores the effect of UWB pulses at the ends of the OFDM symbol interval. Solving for the pulse amplitude, we have

$$C_{R1} = \sqrt{2 \cdot \frac{65}{96Kr} \cdot \frac{T_u}{T_s} \left( \frac{E_b}{N_I} \right)_{\text{target}}^{-1}} \quad (4.4b)$$

The amplitude of an individual pulse (after receiver filtering) at a particular sample time is inversely proportional to the difference between the sample time and the pulse time; as illustrated in Figure 4.2.1, for simulation purposes, we assume that a given OFDM sample is overlapped by filtered UWB pulses arriving in the previous three sample intervals and in the following three sample intervals.

Using (4.4b) to set the desired value of  $E_b/N_I$ , simulations of 802.11a OFDM were made under different levels of UWB pulse interference and Gaussian noise. For the simulation, it was assumed that carrier phase offset is  $\varphi = \pi/4$  and the UWB pulse period is  $T_u = 10$  ns. Results of the simulation for no coding are shown in Figure 4.2.2 for BPSK and QPSK, in Figure 4.2.3 for 16-QAM, and in Figure 4.2.4 for 64-QAM.

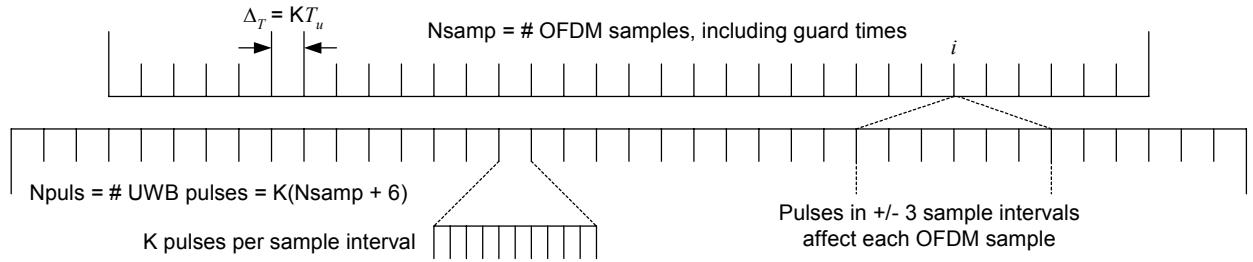


Figure 4.2.1 Relationships among UWB pulses and OFDM samples.

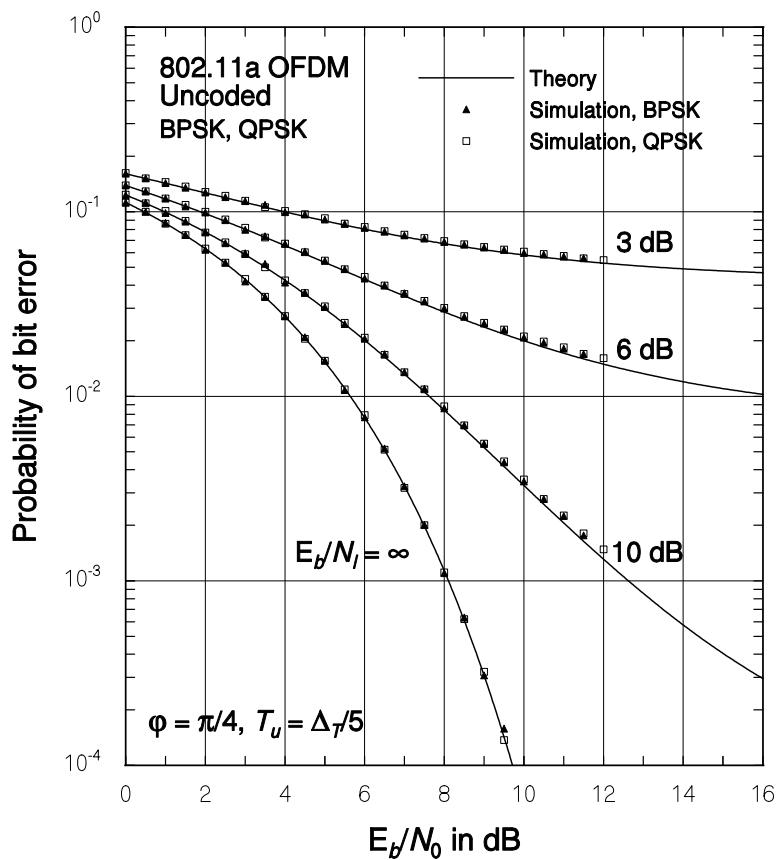


Figure 4.2.2 Comparison of theoretical and simulation results for 802.11a OFDM with BPSK or QPSK and UWB pulse interference at a 100 MHz pulse rate.

It is evident in Figures 4.2.2 and 4.2.3 that the expression used to estimate the UWB power at the FFT output, on the right side of equation (4.4a), underestimates the number of overlapping UWB pulses slightly. The effect is that when the simulation power is set by (4.4b), it is high by less than 0.5 dB—6 samples in 160 samples for a two-symbol packet. In Figure 4.2.4, in addition to this effect we see the error in approximating the BER of 64-QAM using one term.

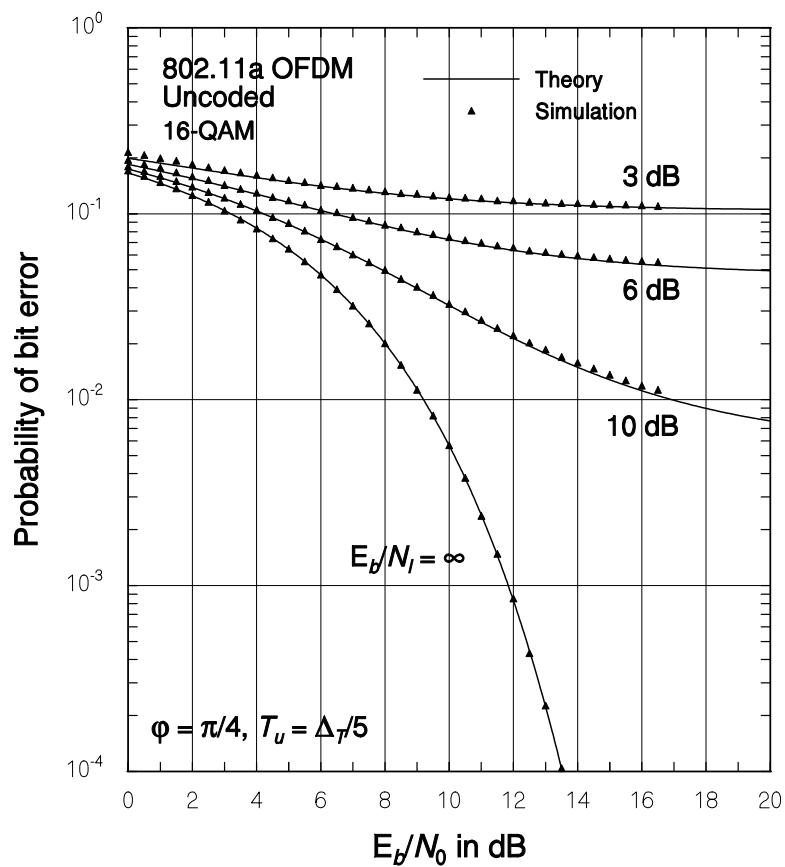


Figure 4.2.3 Comparison of theoretical and simulation results for 802.11a OFDM with 16-QAM and UWB pulse interference at a 100 MHz pulse rate.

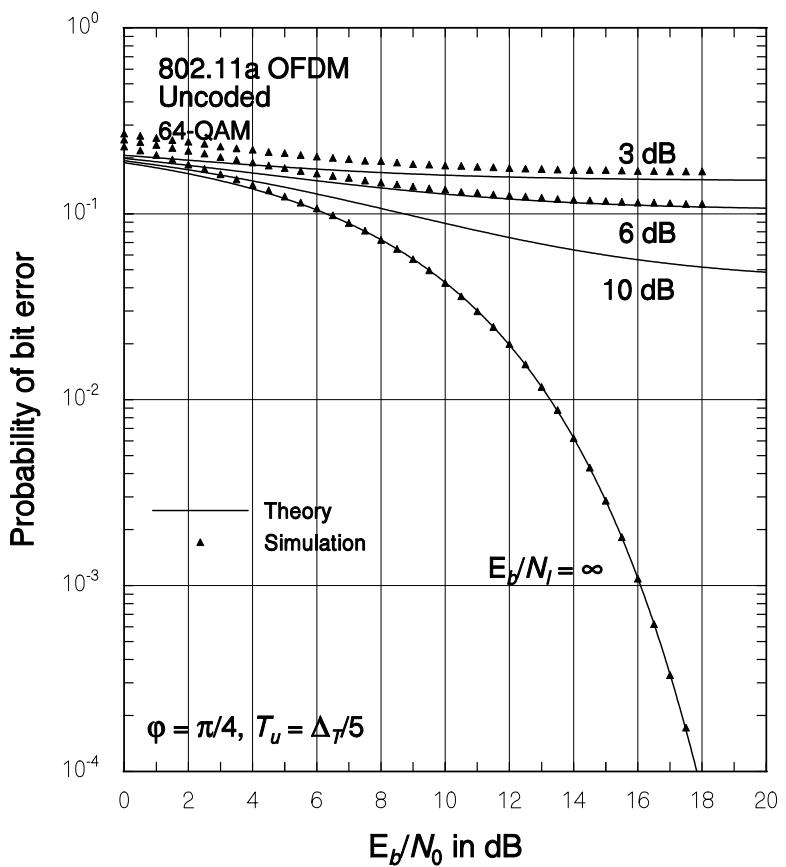


Figure 4.2.4 Comparison of theoretical and simulation results for 802.11a OFDM with 64-QAM and UWB pulse interference at a 100 MHz pulse rate.

## APPENDICES

### A. Mapping of Binary Data to Modulation Symbols

The  $(0, 1)$  binary data out of the encoder are interleaved, then mapped into complex samples of modulation symbols as shown in Figure A.1 to achieve different bit rates. In the text above, the samples are denoted  $a_k = a(kT)$ . Let the sequence of interleaver output binary data bits be denoted  $\{\alpha_i\}$ .

For the case of BPSK modulation, the  $(0,1)$  binary data symbols are mapped to  $(\pm 1 + j0)$  modulation symbols. The number of modulation symbols equals the number of encoder output data bits, and the mapping can be expressed mathematically by

$$\begin{aligned}\operatorname{Re}\{a_k\} &= 2\alpha_k - 1 \\ \operatorname{Im}\{a_k\} &= 0\end{aligned}\tag{A.1}$$

For the case of QPSK modulation, the  $(0,1)$  binary data symbols are mapped to  $\pm 1 + j\pm 1$  modulation symbols. The number of modulation symbols equals half the number of encoder output data bits, and the mapping can be expressed mathematically by

$$\begin{aligned}\operatorname{Re}\{a_k\} &= 2\alpha_{2k} - 1 \\ \operatorname{Im}\{a_k\} &= 2\alpha_{2k+1} - 1\end{aligned}\tag{A.2}$$

For the case of 16-QAM modulation, the  $(0,1)$  binary data symbols are mapped to  $(\pm 1, \pm 3) + j(\pm 1, \pm 3)$  modulation symbols. The number of modulation symbols equals one-fourth the number of encoder output data bits, and the mapping can be expressed mathematically by

$$\begin{aligned}\operatorname{Re}\{a_k\} &= 6\alpha_{4k} + 2\alpha_{4k+1} - 3 - 4\alpha_{4k}\alpha_{4k+1} \\ \operatorname{Im}\{a_k\} &= 6\alpha_{4k+2} + 2\alpha_{4k+3} - 3 - 4\alpha_{4k+2}\alpha_{4k+3}\end{aligned}\tag{A.3}$$

For the case of 64-QAM modulation, the  $(0,1)$  binary data symbols are mapped to  $(\pm 1, \pm 3, \pm 5, \pm 7) + j(\pm 1, \pm 3, \pm 5, \pm 7)$  modulation symbols. The number of modulation symbols equals one-sixth the number of encoder output data bits, and the mapping can be expressed mathematically by

$$\begin{aligned}\operatorname{Re}\{a_k\} &= 14\alpha_{6k} + 6\alpha_{6k+1} + 2\alpha_{6k+2} - 7 - 12\alpha_{6k}\alpha_{6k+1} - 4\alpha_{6k}\alpha_{6k+2} - 4\alpha_{6k+1}\alpha_{6k+2} + 8\alpha_{6k}\alpha_{6k+1}\alpha_{6k+2} \\ \operatorname{Im}\{a_k\} &= 14\alpha_{6k+3} + 6\alpha_{6k+4} + 2\alpha_{6k+5} - 7 - 12\alpha_{6k+3}\alpha_{6k+4} - 4\alpha_{6k+3}\alpha_{6k+5} - 4\alpha_{6k+4}\alpha_{6k+5} + 8\alpha_{6k+3}\alpha_{6k+4}\alpha_{6k+5}\end{aligned}\tag{A.4}$$

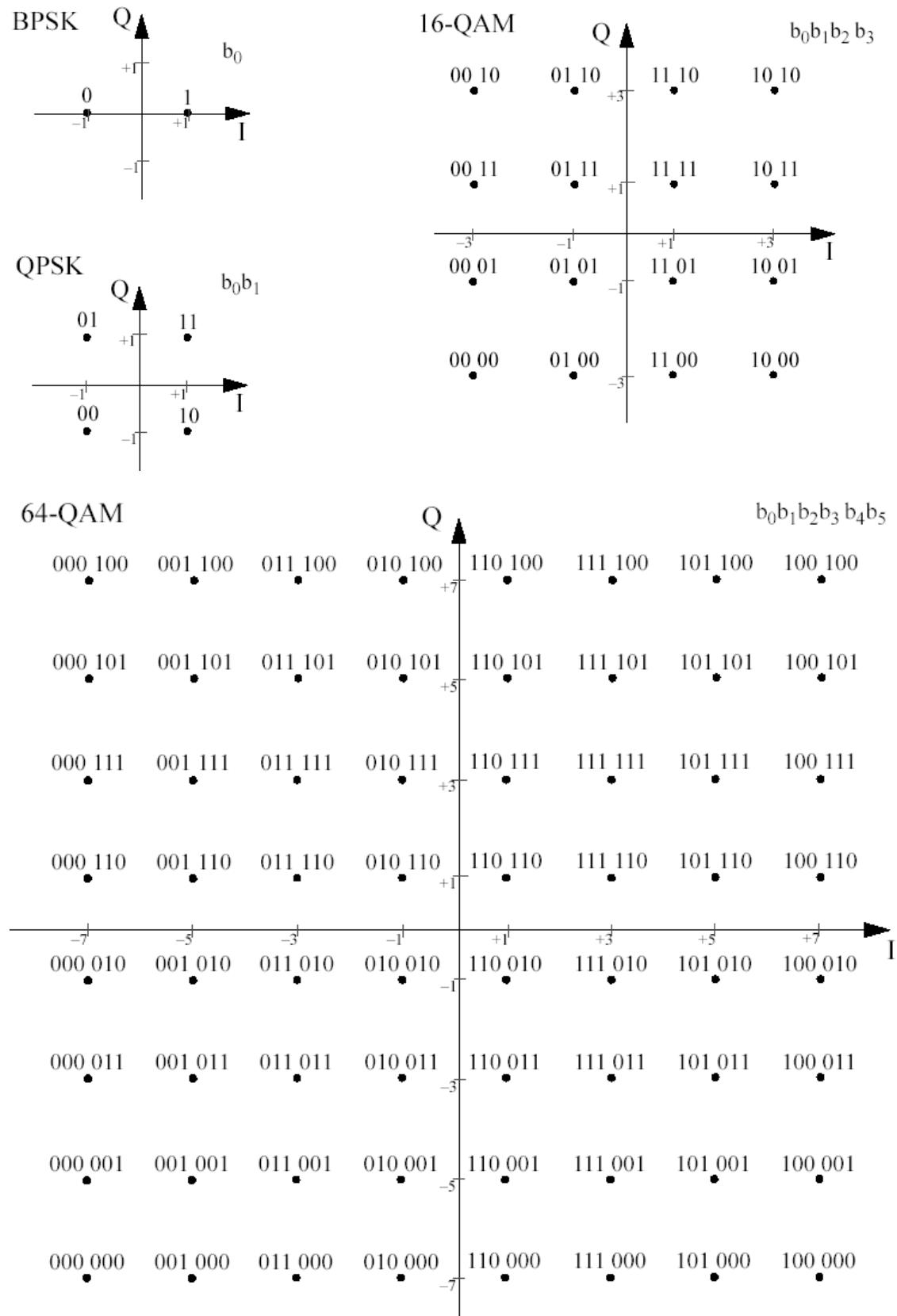


Figure A.1 802.11 Modulation symbol mappings

## B. Calculation of Received OFDM Channel Noise Powers

From the definition of the receiver FFT, we have

$$\begin{aligned} 2\sigma_{ni}^2 &= \text{E}\left\{\left|\mathcal{N}_{ni}\right|^2\right\} = \frac{1}{N_p^2} \sum_{m=0}^{N_p-1} \sum_{r=0}^{N_p-1} \left[ \text{E}\left\{v_{mc} v_{rc}\right\} + \text{E}\left\{v_{ms} v_{rs}\right\} \right] e^{-j2\pi(m-r)i/N_p} \\ &= \frac{2}{N_p^2} \sum_{m=0}^{N_p-1} \sum_{r=0}^{N_p-1} \text{E}\left\{v_{mc} v_{rc}\right\} e^{-j2\pi(m-r)i/N_p} \end{aligned} \quad (\text{B.1})$$

### B.1 Independent Noise Samples

If the noise samples are assumed to be independent, (B.1) leads to the expression

$$\sigma_{ni}^2 = \frac{1}{N_p^2} \sum_{r=0}^{N_p-1} \text{E}\left\{v_{rc}^2\right\} = \frac{1}{N_p} \text{E}\left\{v_{rc}^2\right\} = \frac{\sigma_v^2}{N_p} = \frac{N_0/\Delta_T}{1/\Delta_T\Delta_F} = N_0\Delta_F \quad (\text{B.2})$$

### B.2 Dependent (Oversampled) Noise Samples

Substituting for the noise correlation function of (2.2.2), and using  $1/N_p = \Delta_T\Delta_F$  and  $T_0 = N_p\Delta_T = 1/\Delta_F$ , we find that

$$\begin{aligned} \sigma_{ni}^2 &= N_0 \int_{-B/2}^{B/2} df \frac{1}{N_p^2} \sum_{m=0}^{N_p-1} \sum_{r=0}^{N_p-1} e^{j2\pi f(m-r)\Delta_T} e^{-j2\pi(m-r)i/N_p} \\ &= N_0 \int_{-B/2}^{B/2} df \frac{1}{N_p^2} \left| \sum_{m=0}^{N_p-1} \left( e^{j2\pi(f-i\Delta_F)\Delta_T} \right)^m \right|^2 \\ &= N_0 \int_{-B/2}^{B/2} df \left[ \frac{\sin[\pi(f-i\Delta_F)T_0]}{N_p \sin[\pi(f-i\Delta_F)T_0 / N_p]} \right]^2 \doteq N_0 \Delta_F \int_{-BT_0/2}^{BT_0/2} dx \left[ \frac{\sin[\pi(x-i)]}{\pi(x-i)} \right]^2 \end{aligned} \quad (\text{B.3})$$

where  $BT_0 = 53$ . The approximation in the last line of the previous equation is very slight for  $N_p = 64$ . The integral expression may be solved in terms of the sine integral,  $\text{Si}(x)$ , as follows [6, §2.642.5]:

$$\begin{aligned} \sigma_{ni}^2 &\doteq \frac{N_0\Delta_F}{\pi} \int_{-(53/2+i)\pi}^{(53/2-i)\pi} du \frac{\sin^2 u}{u^2} = \frac{N_0\Delta_F}{\pi} \left[ \text{Si}(2u) - \frac{\sin^2 u}{u} \right]_{-(53/2+i)\pi}^{(53/2-i)\pi} \\ &= \frac{N_0\Delta_F}{\pi} \left[ \text{Si}(53\pi - 2i\pi) + \text{Si}(53\pi + 2i\pi) - \frac{2}{53\pi - 2i\pi} - \frac{2}{53\pi + 2i\pi} \right] \end{aligned} \quad (\text{B.3a})$$

where [7, Table 5.3]

$$\text{Si}(53\pi - 2i\pi) \doteq \begin{cases} \frac{\pi}{2} + \frac{1}{53\pi - 2i\pi}, & i \leq 26 \\ -\frac{\pi}{2} + \frac{1}{53\pi - 2i\pi}, & i > 26 \end{cases} \quad (\text{B.3b})$$

and

$$\text{Si}(53\pi + 2i\pi) \doteq \begin{cases} \frac{\pi}{2} + \frac{1}{53\pi + 2i\pi}, & i \geq -26 \\ -\frac{\pi}{2} + \frac{1}{53\pi + 2i\pi}, & i < -26 \end{cases} \quad (\text{B.3c})$$

Combining (B.3a), (B.3b), and (B.3c), we have

$$\sigma_{ni}^2 \doteq N_0 \Delta_F \times \begin{cases} 1 - \frac{2}{\pi^2} \frac{53}{(53)^2 - (2i)^2}, & |i| \leq 26 \\ -\frac{2}{\pi^2} \frac{53}{(53)^2 - (2i)^2}, & |i| > 26 \end{cases} \quad (\text{B.4})$$

### C. Simulation Results

Table C.1 Numbers of OFDM symbols and bits used for the simulation without coding, with the BER results shown in Figure 4.1.1

Modulation	# packets	# OFDM symbols	# bits
BPSK	1,000, $E_b/N_0 \leq 6$ dB	2,000	80,000
	5,000, $6 < E_b/N_0 \leq 9$ dB	10,000	400,000
	10,000, $9 < E_b/N_0 \leq 11$ dB	20,000	800,000
QPSK	1,000, $E_b/N_0 \leq 8$ dB	2,000	176,000
	2,000, $8 < E_b/N_0 \leq 11$ dB	4,000	352,000
16-QAM	1,000, $E_b/N_0 \leq 8$ dB	2,000	368,000
	2,000, $8 < E_b/N_0 \leq 15$ dB	4,000	736,000
64-QAM	1,000, $E_b/N_0 \leq 16$ dB	2,000	560,000
	2,000, $16 < E_b/N_0 \leq 20$ dB	4,000	1,120,000

Table C.1 Simulation Data for 802.11a OFDM/BPSK and OFDM/QPSK with Coding. Each packet consists of two OFDM symbols, and six encoder tail bits are used.

$E_b/N_0$ (dB)	BPSK								QPSK							
	$r = 1/2$ , soft		$r = 3/4$ , soft		$r = 1/2$ , hard		$r = 3/4$ , hard		$r = 1/2$ , soft		$r = 3/4$ , soft		$r = 1/2$ , hard		$r = 3/4$ , hard	
	# pkt	BER														
0.0	10,000	2.00E-1	5,000	3.21E-1	10,000	3.49E-1	10,000	3.76E-1	5,000	2.78E-1	5,000	4.10E-1	5,000	4.06E-1	5,000	4.51E-1
0.5		1.49E-1		2.88E-1		3.13E-1		3.60E-1		2.13E-1		3.81E-1		3.77E-1		4.36E-1
1.0		1.01E-1		2.42E-1		2.75E-1		3.38E-1		1.51E-1		3.38E-1		3.42E-1		4.20E-1
1.5	20,000	6.06E-2		1.90E-1		2.28E-1		3.09E-1	10,000	9.22E-2		2.81E-1		2.94E-1		3.92E-1
2.0		3.31E-2		1.31E-1		1.80E-1		2.73E-1		4.88E-2		2.08E-1		2.38E-1		3.54E-1
2.5	50,000	1.50E-2	10,000	8.21E-2		1.32E-1		2.27E-1	25,000	2.22E-2	10,000	1.34E-1		1.77E-1		3.10E-1
3.0		6.04E-3		4.05E-2		8.80E-2		1.79E-1		8.54E-3		7.14E-2		1.22E-1		2.53E-1
3.5	100,000	2.13E-3		1.91E-2		5.30E-2		1.28E-1	50,000	2.71E-3		3.05E-2		7.43E-2		1.89E-1
4.0		6.23E-4		6.35E-3		2.91E-2		8.30E-2	100,000	7.38E-4		1.09E-2		4.03E-2		1.26E-1
4.5		1.41E-4	20,000	2.20E-3	20,000	1.41E-2	20,000	4.93E-2		1.78E-4	20,000	3.12E-3		1.78E-2		7.53E-2
5.0		4.21E-5		5.31E-4		6.18E-3		2.60E-2		4.00E-5		7.31E-4		8.40E-3		3.97E-2
5.5		6.43E-6		1.69E-4		2.73E-3		1.31E-2		8.56E-6		1.03E-4	10,000	3.11E-3		1.78E-2
6.0		7.14E-7	100,000	2.80E-5		9.80E-4		4.97E-3		2.33E-7	100,000	3.62E-5		1.27E-3		7.41E-3
6.5				6.21E-6		2.49E-4		1.91E-3				1.06E-5	20,000	2.71E-4	10,000	2.50E-3
7.0				6.06E-7	50,000	8.00E-5	50,000	5.89E-4				1.59E-6		6.83E-5	20,000	7.74E-4
7.5						8.57E-6		2.07E-4					50,000	2.16E-5		1.59E-4
8.0						100,000	5.48E-6	100,000	4.52E-5					5.11E-6		4.09E-5
8.5							2.62E-6		1.02E-5				100,000	4.44E-7	50,000	9.57E-6
9.0																4.64E-6
9.5																
10.0																

Table C.2 Simulation Data for 802.11a OFDM/16-QAM and OFDM/64-QAM with Coding

$E_b/N_0$ (dB)	16-QAM								64-QAM								
	$r = 1/2$ , soft		$r = 3/4$ , soft		$r = 1/2$ , hard		$r = 3/4$ , hard		$r = 2/3$ , soft		$r = 3/4$ , soft		$r = 2/3$ , hard		$r = 3/4$ , hard		
	# pkt	BER															
0.0	5,000	4.39E-1			5,000	4.78E-1	5,000	4.90E-1	2,000	4.92E-1	2,000	4.93E-1					
0.5		4.22E-1				4.73E-1		4.87E-1		4.90E-1		4.92E-1					
1.0		3.98E-1	10,000	4.74E-1		4.67E-1		4.86E-1		4.88E-1		4.91E-1					
1.5		3.62E-1		4.68E-1		4.59E-1		4.84E-1		4.86E-1		4.89E-1					
2.0		3.13E-1		4.60E-1		4.44E-1		4.79E-1		4.84E-1		4.90E-1					
2.5		2.53E-1		4.47E-1		4.26E-1		4.74E-1		4.83E-1		4.87E-1	2,000	4.91E-1			
3.0		1.83E-1		4.29E-1		4.00E-1		4.65E-1		4.77E-1		4.83E-1		4.89E-1			
3.5	10,000	1.17E-1		4.01E-1		3.68E-1		4.55E-1		4.70E-1		4.81E-1		4.89E-1			
4.0		6.31E-2	20,000	3.60E-1		3.25E-1		4.39E-1		4.62E-1		4.77E-1		4.85E-1			
4.5		2.89E-2		3.03E-1	10,000	2.71E-1		4.16E-1	5,000	4.49E-1		4.70E-1		4.80E-1			
5.0	50,000	1.05E-2		2.28E-1		2.12E-1		3.84E-1		4.34E-1		4.65E-1		4.74E-1			
5.5		3.50E-3		1.49E-1		1.53E-1	10,000	3.48E-1		4.09E-1	5,000	4.56E-1		4.68E-1			
6.0	100,000	9.36E-4	50,000	7.94E-2	20,000	1.00E-1		2.95E-1		3.74E-1		4.40E-1		4.59E-1			
6.5		2.19E-4		3.39E-2		6.04E-2		2.36E-1	10,000	3.20E-1		4.19E-1		4.43E-1			
7.0		4.90E-5		1.24E-2		3.32E-2		1.73E-1		2.54E-1		3.89E-1		4.25E-1	5,000	4.55E-1	
7.5	50,000	1.13E-5		3.70E-3		1.59E-2	20,000	1.14E-1		1.75E-1	10,000	3.44E-1		3.98E-1		4.39E-1	
8.0		1.51E-6	100,000	1.02E-3	50,000	7.02E-3		6.76E-2		1.03E-1		2.81E-1		3.65E-1		4.20E-1	
8.5				2.68E-4		2.91E-3		3.51E-2	20,000	5.07E-2		2.04E-1		3.23E-1		3.92E-1	
9.0			200,000	6.64E-5	100,000	1.09E-3		1.66E-2		2.02E-2		1.26E-1		2.76E-1		3.59E-1	
9.5			50,000	1.62E-5		3.77E-4	50,000	7.01E-3		6.57E-3	20,000	6.36E-2		2.17E-1		3.15E-1	
10.0				2.34E-6		1.15E-4		2.69E-3		2.07E-3		2.66E-2		1.64E-1		2.59E-1	
10.5						3.87E-5	100,000	9.19E-4	50,000	5.31E-4		9.11E-3	5,000	1.10E-1	10,000	2.03E-1	
11.0						1.07E-5		2.76E-4		1.40E-4		3.00E-3		6.85E-2		1.44E-1	
11.5						1.34E-6		8.15E-5		3.50E-5	100,000	8.24E-4		3.78E-2		9.38E-2	
12.0								1.80E-5	100,000	6.93E-6		2.35E-4		2.00E-2		5.57E-2	
12.5								200,000	3.05E-6		1.19E-6		6.67E-5		9.21E-3	3.01E-2	
13.0								100,000	1.13E-6		3.17E-7		1.97E-5	10,000	3.94E-3	20,000	1.43E-2
13.5									6.38E-7				5.80E-6		1.67E-3	6.42E-3	
14.0													1.08E-6		6.18E-4		2.57E-3
14.5															1.90E-4	50,000	9.37E-4
15.0														50,000	5.32E-5		3.30E-4
15.5															1.55E-5		9.77E-5
16.0														100,000	3.86E-6		2.85E-5
16.5															8.47E-7	100,000	6.50E-6
17.0															3.70E-7		2.25E-6
17.5																3.52E-7	

## D. SIMULATION PROGRAM

Listed below are the source files for the 802.11a/UWB coexistence simulation, which were developed and run in Microsoft C++ 6.0. The files included are the following:

ofdm.h	ofdm.cpp
Channel.cpp	Rx.cpp
ConvCode.cpp	RxSoft.cpp
main.cpp	Tx.cpp

### **File ofdm.h**

This file is the header file for the simulation.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define PI 3.1415926535897932
#define FFTLength 64.0//8192.0//256.0///16384.0
#define GrdLength ((int)FFTLength/4)
#define TtlLength ((int)FFTLength+GrdLength+1)
#define Packbuff 16*TtlLength
#define PackBlock 48
#define IB1 1
#define IB3 4
#define IB20 524288
#define MASK IB3
#define NormQPSK 1/sqrt(2)
#define Norm16QAM 1/sqrt(10)
#define Norm64QAM 1/sqrt(42)
#define HARD 1
#define SOFT 0
#define SampleTime (4e-6/(TtlLength))
#define FilterSize (int)(50e-9/(SampleTime))
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
#define YES "Yes"

extern long      x1,x2,x3,x4,x5,x6;
extern int       x10,x20,x30,x40,x50,x60;
extern unsigned int iseed;
extern short packet[20*PackBlock],ppacket[20*PackBlock/16], decod[20*PackBlock/16];
extern double Inph[Packbuff], Quad[Packbuff];
extern double Inphi[Packbuff], QuadI[Packbuff];
extern double Inpho[Packbuff], Quado[Packbuff];
extern short int data[20*PackBlock];
extern double dataf[20*PackBlock];
extern char Branch[128];
extern double CI;                                // Global power calibration factor

extern char *Bpsk;
extern char *Qpsk;
extern char *Qam16;
extern char *Qam64;

void trellis (void);
void UwbInt2( double Tu, double Tf, int palen, double pamp, double phi, int zero);
int   BpskTx(int palen, char *bypassconv, char *rate);
int   QpskTx(int palen, char *bypassconv, char *rate);
int   QAM16Tx(int palen, char *bypassconv, char *rate);
```

```

int QAM64Tx(int palen, char *bypassconv, char *rate);
int BpskRx(int palen, char *bypassconv, char *rate, char *decision);
int QpskRx(int palen, char *bypassconv, char *rate, char *decision);
int QAM16Rx(int palen, char *bypassconv, char *rate, char *decision);
int QAM64Rx(int palen, char *bypassconv, char *rate, char *decision);
int DeConv12(int palen, char *bypassconv);
int DeConv23(int palen, char *bypassconv);
int DeConv34(int palen, char *bypassconv);
int DeConv12Soft(int palen);
int DeConv23Soft(int palen);
int DeConv34Soft(int palen);
void AwgnChannel (double n0, int palen, int zero);
int Error(int palen);
void Write( char *str, char *bypass, char *rate, char *decision, int N1, int N2, int N);
void BPSKDemSoft(int palen);
int QPSKDermSoft(int palen);
int QAM16DemSoft(int palen);
int QAM64DemSoft(int palen);
void fourl(double ddata[], unsigned int nn, int isign);
int LengthCal(char *Mod,char *Bypass, char *Rate);
double hoft(double t, double a);

```

## File Channel.cpp

This file contains the functions and procedures that generate the noise samples and the UWB interference samples to add to the baseband received OFDM signal.

```

#include "ofdm.h"

//*****************************************************************************
// This function generates the coefficients of the root-squared raised cosine
// filter. The input is 'a' as roll-off factor and the output is written in
// array 'htp'.
//**************************************************************************

void RaisedCosine(double *htp, double a){

    int i;
    for ( i = 0; i < 6*FilterSize ; i++){
        double temp= i - 3*FilterSize ;
        temp /= FilterSize;
        if((temp) && (1-16*temp*temp*a*a))
            htp[i]= ( sin(PI*(1-a)*temp) + 4*a*temp*cos (PI*(1+a)*temp) )
            /((PI*temp) * (1-16*temp*temp*a*a));
        if(!temp)
            htp[i] = 1-a+4*a/PI;
        if(!(1-16*temp*temp*a*a)){
            htp[i]= (a/sqrt(2.))*((1+2/PI)*sin(PI/(4*a))+(1-2/PI)*cos(PI/(4*a)));
        }
    } // for (int i = 0; i < TDELAY; i+
}

//*****************************************************************************
// Function to calculate individual values of the filter impulse response.
//**************************************************************************

double hoft(double t, double a){

    double temp = 0.;
    if((t) && (1-16.*t*t*a*a))
        temp = ( sin(PI*(1-a)*t) + 4.*a*t*cos (PI*(1+a)*t) )
        /((PI*t) * (1-16.*t*t*a*a));
    if(!t)
        temp = 1-a+4.*a/PI;
    if(!(1-16*t*t*a*a))
        temp = (a/sqrt(2.))*((1+2./PI)*sin(PI/(4.*a))+(1-2./PI)*cos(PI/(4.*a)));
    return (temp);
}

```

```

}

***** This function generates the guassian noise samples and add them to the
***** input samples located in global Inpho and Quado variables. The power of the
***** noise is determined by varn. palen determines the length of the packet and
***** zero can bypass this function for calibration purposes.
***** */

void NoiseGen(double varn, int palen, int zero){
    double m1,m2;
    double NI,NQ;
    palen=palen*TtlLength/48;
    if(zero){
        for(int j=0; j<palen; j++){
            x1 = (171 * x1)%30269;
            x2 = (172 * x2)%30307;
            x3 = (170 * x3)%30323;
            m1 = fmod( (double(x1)/30269. + double(x2)/30307. + double(x3)/30323.), 1.);
            // m1 is a uniform rv
            x4 = (171 * x4)%30269;
            x5 = (172 * x5)%30307;
            x6 = (170 * x6)%30323;
            m2 = fmod( (double(x4)/30269. + double(x5)/30307. + double(x6)/30323.), 1.);
            // m2 is a uniform rv
            NI=sqrt(-2*varn*log(m1))*cos(PI*2*m2);
            NQ=sqrt(-2*varn*log(m1))*sin(PI*2*m2);
            Inpho[j] += NI ;
            Quado[j] += NQ ;
        } // for(j=0; j<Ns*16*palen; j++)
    } //zero
    else{
        for(int j=0; j<palen; j++){
            x1 = (171 * x1)%30269;
            x2 = (172 * x2)%30307;
            x3 = (170 * x3)%30323;
            m1 = fmod( (double(x1)/30269. + double(x2)/30307. + double(x3)/30323.), 1.);
            // m1 is a uniform rv
            x4 = (171 * x4)%30269;
            x5 = (172 * x5)%30307;
            x6 = (170 * x6)%30323;
            m2 = fmod( (double(x4)/30269. + double(x5)/30307. + double(x6)/30323.), 1.);
            // m2 is a uniform rv
            NI=sqrt(-2*varn*log(m1))*cos(PI*2*m2);
            NQ=sqrt(-2*varn*log(m1))*sin(PI*2*m2);
            Inpho[j] = NI ;
            Quado[j] = NQ ;
        } // for(j=0; j<Ns*16*palen; j++)
    } //zero
}

***** This function simulates AWGN channel by calling NoiseGen which adds
***** noise to the input.
***** */

void AwgnChannel (double varn, int palen, int zero){

    NoiseGen(varn, palen, zero);
}

***** This function generates samples of the uwb interference signal in global
***** 'InphI' and 'Quadi' arrays and then adds them to the signal array in
***** 'Inpho' and 'Quado'. The input is the uwb pulse period Tu and effective
***** pulse amplitude pamp, and the inverse of the filter bandwidth Tf.
***** 'zero' is for calibration purposes.
***** */

void UwbInt2( double Tu, double Tf, int palen, double pamp, double phi, int zero){

    int i,j,Nsamp,Npuls,K,L;
}

```

```

double dd1,d1[1000],delt1,delt2;
Nsamp = palen * TtlLength / 48;
pulen = Nsamp;
K = (int)((3.2e-6)/FFTLength/Tu);
Npuls = (Nsamp + 6)*K + 1;
delt1 = (3.2e-6)/FFTLength/Tf;
delt2 = Tu/Tf;
L = RAND_MAX;
for(i=0;i < Npuls; i++){ // Generate UWB pulse modulation data
    dd1 = (double)(rand())/L;
    if (dd1 > .5)
        d1[i] = 1.0*pamp;
    else
        d1[i] = -1.0*pamp;
}
for (i=0;i < Nsamp; i++){ //Initialize UWB baseband samples
    InphI[i] = 0;
    QuadI[i] = 0;
}
for(i=0;i < Nsamp; i++){ //Calculate UWB baseband samples
    for(j=0; j<6*K; j++){ //Add overlapping pulses, weighted by hoft
        InphI[i] += (cos(phi)) * d1[j+K*i] * hoft((3*K-j)*delt2, 0.0);
        QuadI[i] += (sin(phi)) * d1[j+K*i] * hoft((3*K-j)*delt2, 0.0);
    }
    if(zero){ //Add UWB samples to signal samples
        for(i=0;i<Nsamp; i++){
            Inpho[i] += InphI[i];
            Quado[i] += QuadI[i];
        }
    }
    else{ //For test, replace signal samples with UWB samples
        for(i=0;i<Nsamp; i++){
            Inpho[i] = InphI[i];
            Quado[i] = QuadI[i];
        }
    }
}
}

```

## File ConvCode.cpp

This file contains the functions and procedures for hard and soft decoding of the demultiplexed OFDM output data.

```

#include "ofdm.h"

// Convolutional decoder, rate 1/2, hard decision making

int DeConv12(int palen, char *bypassconv){

    unsigned short int Old[64]={0}, New[64];
    unsigned short static int NewPath[64][Packbuff], OldPath[64][Packbuff];
    unsigned short int A,B;
    int i,j,k;
    static int sw=1;
    if (!strcmp(bypassconv,"No")){
        if (sw){
            printf("\n 1/2 rate convolutional code, HARD Decision");
            printf("\n Using a priori information on tail bits");
        }
        sw=0;
        for (i=1; i<64; i++)
            Old[i]=100;
        for (i=0;i<pulen; i+=2){
            for (j=0;j<32; j++){
                A=Old[2*j]+(data[i]^((Branch[4*j]&0x1)))+ (data[i+1]^((Branch[4*j]>>1)));

```

```

B=Old[2*j+1]+(data[i]^(Branch[4*j+1]&0x1))+(data[i+1]^(Branch[4*j+1]>>1));
if (A<B){
    New[j]=A;
    for ( k=0; k<i/2;k++)
        NewPath[j][k] = OldPath[2*j][k];
    NewPath[j][i/2]=0;
}
else{
    New[j]=B;
    for ( k=0; k<i/2;k++)
        NewPath[j][k] = OldPath[2*j+1][k];
    NewPath[j][i/2]=0;
}
A=Old[2*j]+(data[i]^(Branch[4*j+2]&0x1))+(data[i+1]^(Branch[4*j+2]>>1));

B=Old[2*j+1]+(data[i]^(Branch[4*j+3]&0x1))+(data[i+1]^(Branch[4*j+3]>>1));
if (A<B){
    New[32+j]=A;
    for ( k=0; k<i/2;k++)
        NewPath[j+32][k] = OldPath[2*j][k];
    NewPath[j+32][i/2]=1;
}
else{
    New[32+j]=B;
    for ( k=0; k<i/2;k++)
        NewPath[j+32][k] = OldPath[2*j+1][k];
    NewPath[j+32][i/2]=1;
}
for (j=0; j<64; j++){
    Old[j]=New[j];
    for(k=0;k<i/2+1;k++){
        OldPath[j][k]=NewPath[j][k];
    }
}
//Use a priori information on tail bits to choose path for state 0
for ( i = 0; i < palen/2; i++)
    data[i] = NewPath[0][i];
return palen/2;
}
return palen;
}

// Convolutional decoder, rate 2/3, hard decision making

int DeConv23(int palen, char *bypassconv){

unsigned short int Old[64]={0}, New[64];
unsigned static short int NewPath[64][Packbuff], OldPath[64][Packbuff];
unsigned short int A,B,br1,br2,br3,br4,d0[Packbuff];
int i,j,k;
static int sw=1;
if (!strcmp(bypassconv,"No")){
    if (sw){
        printf("\n 2/3 rate convolutional code, HARD Decision");
        printf("\n Using a priori information on tail bits");
    }
    sw=0;
}

for (i=0;i<palen/3; i++){
    d0[4*i]=data[3*i];
    d0[4*i+1]=data[3*i+1];
    d0[4*i+2]=data[3*i+2];
    d0[4*i+3]=3000;
}
for (i=1; i<64; i++)
    Old[i]=100;
for (i=0;i<(palen*2/3)*2; i+=2){
    for (j=0;j<32; j++){
        if (i%4==0){

```

```

        br1=(d0[i]^(Branch[4*j]&0x1)) + (d0[i+1]^( (Branch[4*j]>>1) ) );
        br2=(d0[i]^(Branch[4*j+1]&0x1))+(d0[i+1]^( (Branch[4*j+1]>>1) ) );
        br3=(d0[i]^(Branch[4*j+2]&0x1))+(d0[i+1]^( (Branch[4*j+2]>>1) ) );
        br4=(d0[i]^(Branch[4*j+3]&0x1))+(d0[i+1]^( (Branch[4*j+3]>>1) ) );
    }
    if (i%4==2){
        br1=( d0[i]^(Branch[4*j]&0x1) );
        br2=( d0[i]^(Branch[4*j+1]&0x1) );
        br3=( d0[i]^(Branch[4*j+2]&0x1) );
        br4=( d0[i]^(Branch[4*j+3]&0x1) );
    }
    A=Old[2*j]+br1;
    B=Old[2*j+1]+br2;
    if (A<B){
        New[j]=A;
        for ( k=0; k<(i/2);k++)
            NewPath[j][k] = OldPath[2*j][k];
        NewPath[j][i/2]=0;
    }
    else{
        New[j]=B;
        for ( k=0; k<(i/2);k++)
            NewPath[j][k] = OldPath[2*j+1][k];
        NewPath[j][i/2]=0;
    }
    A=Old[2*j]+br3;
    B=Old[2*j+1]+br4;
    if (A<B){
        New[32+j]=A;
        for ( k=0; k<(i/2);k++)
            NewPath[j+32][k] = OldPath[2*j][k];
        NewPath[j+32][i/2]=1;
    }
    else{
        New[32+j]=B;
        for ( k=0; k<(i/2);k++)
            NewPath[j+32][k] = OldPath[2*j+1][k];
        NewPath[j+32][i/2]=1;
    }
    for (j=0; j<64; j++){
        Old[j]=New[j];
        for(k=0;k<(i/2)+1;k++){
            OldPath[j][k]=NewPath[j][k];
        }
    }
}
//Using a priori information on tail bits to choose path into state 0
for ( i = 0; i < palen*2/3; i++)
    data[i] = NewPath[0][i];

return palen*2/3;
}
return palen;
}

// Convolutional decoder, rate 3/4, hard decision making

int DeConv34(int palen, char *bypassconv){

    unsigned short int Old[64]={0}, New[64];
    unsigned short static int NewPath[64][Packbuff], OldPath[64][Packbuff];
    unsigned short int A,B,b1,b2,b3,b4,d0[Packbuff];
    int i,j,k;
    static int sw=1;
    if (!strcmp(bypassconv,"No")){
        if (sw){
            printf("\n 3/4 rate convolutional code, HARD Decision");
            printf("\n Using a priori information on tail bits");
        }
        sw=0;
    }
}

```

```

for (i=0;i<palen/4; i++){
    d0[6*i]=data[4*i];
    d0[6*i+1]=data[4*i+1];
    d0[6*i+2]=data[4*i+2];
    d0[6*i+3]=3000;
    d0[6*i+4]=3000;
    d0[6*i+5]=data[4*i+3];
}
for (i=1; i<64; i++)
    Old[i]=100;
for (i=0;i<palen*3/4*2; i+=2){
    for (j=0; j<32; j++){
        if (i%6==0){
            br1=(d0[i]^(Branch[4*j]&0x1)) + (d0[i+1]^( (Branch[4*j]>>1) ) );
            br2=(d0[i]^(Branch[4*j+1]&0x1))+(d0[i+1]^( (Branch[4*j+1]>>1) ) );
            br3=(d0[i]^(Branch[4*j+2]&0x1))+(d0[i+1]^( (Branch[4*j+2]>>1) ) );
            br4=(d0[i]^(Branch[4*j+3]&0x1))+(d0[i+1]^( (Branch[4*j+3]>>1) ) );
        }
        if (i%6==2){
            br1=( d0[i]^(Branch[4*j]&0x1) );
            br2=( d0[i]^(Branch[4*j+1]&0x1) );
            br3=( d0[i]^(Branch[4*j+2]&0x1) );
            br4=( d0[i]^(Branch[4*j+3]&0x1) );
        }
        if (i%6==4){
            br1= (d0[i+1]^(Branch[4*j]>>1) );
            br2= (d0[i+1]^(Branch[4*j+1]>>1) );
            br3= (d0[i+1]^(Branch[4*j+2]>>1) );
            br4= (d0[i+1]^(Branch[4*j+3]>>1) );
        }
        A=Old[2*j]+br1;
        B=Old[2*j+1]+br2;
        if (A<B){
            New[j]=A;
            for ( k=0; k<i/2;k++)
                NewPath[j][k] = OldPath[2*j][k];
            NewPath[j][i/2]=0;
        }
        else{
            New[j]=B;
            for ( k=0; k<i/2;k++)
                NewPath[j][k] = OldPath[2*j+1][k];
            NewPath[j][i/2]=0;
        }
        A=Old[2*j]+br3;
        B=Old[2*j+1]+br4;
        if (A<B){
            New[32+j]=A;
            for ( k=0; k<i/2;k++)
                NewPath[j+32][k] = OldPath[2*j][k];
            NewPath[j+32][i/2]=1;
        }
        else{
            New[32+j]=B;
            for ( k=0; k<i/2;k++)
                NewPath[j+32][k] = OldPath[2*j+1][k];
            NewPath[j+32][i/2]=1;
        }
        for (j=0; j<64; j++){
            Old[j]=New[j];
            for(k=0;k<i/2+1;k++){
                OldPath[j][k]=NewPath[j][k];
            }
        }
    }
}

// A priori information on tail bits used to choose state 0 path
for ( i = 0; i < palen*3/4; i++)
    data[i] = NewPath[0][i];

return palen*3/4;

```

```

        }
    return palen;
}

// Convolutional decoder, rate 1/2, soft decision making

int DeConv12Soft(int palen){
    double Old[64]={0}, New[64];
    unsigned short static int newPath[64][Packbuff], OldPath[64][Packbuff];
    double A,B;
    int i,j,k;
    double a1,a2,b1,b2;
    static int sw=1;
    if (sw){
        printf("\n 1/2 rate convolutional code, Soft Decision");
        printf("\n Using a priori information on tail bits" );
    }
    sw=0;

    for (i=0; i<64; i++)
        Old[i]=-100;
    Old[0]=0;
    for (i=0;i<palen; i+=2){
        for (j=0;j<32; j++){
            a1=2*( Branch[4*j]&0x1 )-1;
            a2=2*( Branch[4*j]>>1 )-1;
            b1=2*( Branch[4*j+1]&0x1 )-1 ;
            b2=2*( Branch[4*j+1]>>1 )-1;
            A=Old[2*j]+dataf[i]*a1 + dataf[i+1]*a2;
            B=Old[2*j+1]+ dataf[i]*b1+ dataf[i+1]*b2;
            if (A>B){
                New[j]=A;
                for ( k=0; k<i/2;k++)
                    newPath[j][k] = OldPath[2*j][k];
                newPath[j][i/2]=0;
            }
            else{
                New[j]=B;
                for ( k=0; k<i/2;k++)
                    newPath[j][k] = OldPath[2*j+1][k];
                newPath[j][i/2]=0;
            }
            A=Old[2*j]+dataf[i]*b1+dataf[i+1]*b2;
            B=Old[2*j+1]+dataf[i]*a1+dataf[i+1]*a2;
            if (A>B){
                New[32+j]=A;
                for ( k=0; k<i/2;k++)
                    newPath[j+32][k] = OldPath[2*j][k];
                newPath[j+32][i/2]=1;
            }
            else{
                New[32+j]=B;
                for ( k=0; k<i/2;k++)
                    newPath[j+32][k] = OldPath[2*j+1][k];
                newPath[j+32][i/2]=1;
            }
        }
        for (j=0; j<64; j++){
            Old[j]=New[j];
            for(k=0;k<i/2+1;k++){
                OldPath[j][k]=newPath[j][k];
            }
        }
    }
    //Using a priori information on the tail bits, choose path for state 0
    for ( i = 0; i < palen/2; i++)
        data[i] = newPath[0][i];
    return palen/2;
}

// Convolutional decoder, rate 2/3, soft decision making

```

```

int DeConv23Soft(int palen){

    double Old[64]={0}, New[64];
    unsigned short static int newPath[64][Packbuff], oldPath[64][Packbuff];
    double A,B,br1,br2,br3,br4,d0[Packbuff],a1,a2,b1,b2;
    int i,j,k;
    static int sw=1;
    if (sw){
        printf("\n 2/3 rate convolutional code, Soft Decision");
        printf("\n Using a priori information on tail bits");
    }
    sw=0;

    for (i=0;i<palen/3; i++){
        d0[4*i]=dataf[3*i];
        d0[4*i+1]=dataf[3*i+1];
        d0[4*i+2]=dataf[3*i+2];
        d0[4*i+3]=3000;
    }
    for (i=1; i<64; i++)
        Old[i]=-100;
    for (i=0;i<palen*2/3*2; i+=2){
        for (j=0;j<32; j++){
            for (j=0;j<32; j++){
                a1=2*( Branch[4*j]&0x1 )-1;
                a2=2*( Branch[4*j]>>1 )-1;
                b1=2*( Branch[4*j+1]&0x1 )-1 ;
                b2=2*( Branch[4*j+1]>>1 )-1;
                if (i%4==0){
                    br1=d0[i]*a1 + d0[i+1]*a2;
                    br2=d0[i]*b1 + d0[i+1]*b2;
                    br3=d0[i]*b1 + d0[i+1]*b2;
                    br4=d0[i]*a1 + d0[i+1]*a2;
                }
                if (i%4==2){
                    br1= d0[i]*a1;
                    br2= d0[i]*b1;
                    br3= d0[i]*b1;
                    br4= d0[i]*a1;
                }
                A=Old[2*j]+br1;
                B=Old[2*j+1]+br2;
                if (A>B){
                    New[j]=A;
                    for ( k=0; k<i/2;k++)
                        newPath[j][k] = oldPath[2*j][k];
                    newPath[j][i/2]=0;
                }
                else{
                    New[j]=B;
                    for ( k=0; k<i/2;k++)
                        newPath[j][k] = oldPath[2*j+1][k];
                    newPath[j][i/2]=0;
                }
                A=Old[2*j]+br3;
                B=Old[2*j+1]+br4;
                if (A>B){
                    New[32+j]=A;
                    for ( k=0; k<i/2;k++)
                        newPath[j+32][k] = oldPath[2*j][k];
                    newPath[j+32][i/2]=1;
                }
                else{
                    New[32+j]=B;
                    for ( k=0; k<i/2;k++)
                        newPath[j+32][k] = oldPath[2*j+1][k];
                    newPath[j+32][i/2]=1;
                }
            }
        for (j=0; j<64; j++){
            Old[j]=New[j];
            for(k=0;k<i/2+1;k++){
                oldPath[j][k]=newPath[j][k];
            }
        }
    }
}

```

```

        }
    }

// Using a priori information on tail bits
for ( i = 0; i < palen*2/3; i++)
    data[i] = NewPath[0][i];

return palen*2/3;
}

// Convolutional decoder, rate 3/4, soft decision making

int DeConv34Soft(int palen){

    double Old[64]={0}, New[64];
    unsigned short static int NewPath[64][Packbuff], OldPath[64][Packbuff];
    double A,B,br1,br2,br3,br4,d0[Packbuff],a1,a2,b1,b2;
    int i,j,k;
    static int sw=1;
    if (sw){
        printf("\n 3/4 rate convolutional code, Soft Decision");
        printf("\n Using a priori information on tail bits");
    }
    sw=0;

    for (i=0;i<palen/4; i++){
        d0[6*i]=dataf[4*i];
        d0[6*i+1]=dataf[4*i+1];
        d0[6*i+2]=dataf[4*i+2];
        d0[6*i+3]=0; //Value of punctured bits is arbitrary
        d0[6*i+4]=0; //since they are not used.
        d0[6*i+5]=dataf[4*i+3];
    }
    for (i=1; i<64; i++)
        Old[i]=-100;
    for (i=0;i<palen*3/4*2; i+=2){ //i indexes the decoder output bits
        for (j=0;j<32; j++){ //j indexed the decoder state in reverse bit order
            a1=2*( Branch[4*j]&0x1 )-1;//A coder output bit for state 2*j
            a2=2*( Branch[4*j]>>1 )-1; //B coder output bit for state 2*j
            b1=2*( Branch[4*j+1]&0x1 )-1 ;
            b2=2*( Branch[4*j+1]>>1 )-1;

            if (i%6==0){
                br1=d0[i]*a1 + d0[i+1]*a2;
                br2=d0[i]*b1 + d0[i+1]*b2;
                br3=d0[i]*b1 + d0[i+1]*b2;
                br4=d0[i]*a1 + d0[i+1]*a2;
            }
            if (i%6==2){ //Punctured bits not used in metric calculation
                br1= d0[i]*a1;
                br2= d0[i]*b1;
                br3= d0[i]*b1;
                br4= d0[i]*a1;
            }
            if (i%6==4) {
                br1= d0[i+1]*a2;
                br2= d0[i+1]*b2;
                br3= d0[i+1]*b2;
                br4= d0[i+1]*a2;
            }
            A=Old[2*j]+br1;
            B=Old[2*j+1]+br2;
            if (A>B){
                New[j]=A;
                for ( k=0; k<i/2;k++)
                    NewPath[j][k] = OldPath[2*j][k];
                NewPath[j][i/2]=0;
            }
            else{
                New[j]=B;
                for ( k=0; k<i/2;k++)

```

```

        NewPath[j][k] = OldPath[2*j+1][k];
        NewPath[j][i/2]=0;
    }
    A=Old[2*j]+br3;
    B=Old[2*j+1]+br4;
    if (A>B){
        New[32+j]=A;
        for ( k=0; k<i/2;k++)
            NewPath[j+32][k] = OldPath[2*j][k];
        NewPath[j+32][i/2]=1;
    }
    else{
        New[32+j]=B;
        for ( k=0; k<i/2;k++)
            NewPath[j+32][k] = OldPath[2*j+1][k];
        NewPath[j+32][i/2]=1;
    }
}
for (j=0; j<64; j++){
    Old[j]=New[j];
    for(k=0;k<i/2+1;k++){
        OldPath[j][k]=NewPath[j][k];
    }
}
//Use a priori information on tail bits to select path to state 0
for ( i = 0; i < palen*3/4; i++)
    data[i] = NewPath[0][i];

return palen*3/4;
}

```

## File main.cpp

This file contains the main program as well as functions to determine the number of binary symbols in the OFDM packets and to calculate the number of bit errors.

```

//PROGRAM to simulate IEEE 802.11a OFDM signals, noise, and interference from an
//UWB pulse train with random antipodal data modulation
//Developed by NIST guest researcher Amir Soltanian, April 2003
//Modified by L. E. Miller, NIST Wireless Communication Technologies Group, September
2003

#include "ofdm.h"

double ber[12][50]; // output: Average BER for different CIR and Eb/No

void main(void){

    int i,k,j;
    double varn, pamp; // Noise variance, effective UWB pulse amplitude
    double EbNodb;
    int er, N1=1, N2=10;// # Errors, # Eb/Nuwb values, # Eb/No values
    int npackets;
    int palen;
    double phi=PI/3.,Ts=3.2e-6, Tf= 5.0e-8, Tu=5.0e-8, EbNuwb;
    // Recvr I/Q LOs phase wrt UWB, OFDM symbol pd, filter inv. BW, Rep. pd of UWB pulses,
    Eb/Nuwb
    clock_t start,finish; double duration;
    srand(0);
    char Mod[80]="Bpsk"; // Modulation type: Bpsk, Qpsk, Qam16, Qam64
    char Bypass[4]="Yes"; // Bypass convolutional coding: Yes, No
    char Decision[10]="Hard"; // Convolutional decoding: Hard,Soft
    char Rate[10]="3/4"; // coding rate: 1/2, 2/3, 3/4
    double RateVal = 0.75;
    if(!strcmp(Bypass, YES)) RateVal = 1.0;

    trellis(); // Generate the trellis for Viterbi decoding

```

```

    palen=LengthCal(Mod, Bypass, Rate); // calculate packet length according to
modulation

    if(!strcmp(Mod, Bpsk)){           // if the modulation is BPSK
        for ( k= 0; k< 2; k++) {      // Loop on EbNuwb
            EbNuwb=6.0 + (k)*4.0;    // Eb/Nuwb in dB
        //}
        // if (k==0)
        //     EbNuwb=5000;           // AWGN condition
        printf("\n EbNuwb = %4.1f dB", EbNuwb);
        pamp = sqrt(2.*Tu*65.0/96.0/RateVal/Ts)*pow(10.,(-.05*EbNuwb)); //UWB pulse
amplitude
        npackets=1000;              // number of packets per one run of simulation
        for ( i= 0; i< 24; i++) {   // Loop on EbNo
            if (i> 8)
                npackets=2000;
            if (i> 13 )
                npackets=5000;
            if (i> 15)
                npackets=10000;
            if (i> 17)
                npackets=20000;
            er=0;
            EbNodb= i* 0.5;         // Eb/N0 in dB
            varn=(65.0/96.0/RateVal/FFTLength)*pow(10.,(-.1*EbNodb)); // calculate
noise variance
            start=clock();
            for ( j= 0; j< npackets; j++) {
                palen=BpskTx(palen, Bypass,Rate); //Baseband Transmitter
                AwgnChannel(varn, palen, 1);        // Add AWGN Noise
                UwbInt2( Tu, Tf, palen, pamp, phi, 1); // Add UWB Interference
                palen=BpskRx(palen,Bypass,Rate,Decision); // Baseband Receiver
                er += Error(palen); // Calculate errors in packet
                i=i;
                } //for j
                ber[k][i] = er/(npackets * (palen - 6.0) );
                printf ("\n # packets=%6.0i EbNo=%4.1f dB, BER = %4.2e", npackets,
EbNodb, ber[k][i]);
                i=i;
            } // for i
            printf("\n");
        } //for k
    } //if
    if(!strcmp(Mod, Qpsk)){ // Qpsk Modulation
        for ( k= 0; k< 1; k++) {
            EbNuwb=6.+ k*4.0;
            if (k==0)
                EbNuwb=100000;
            printf("\n EbNuwb = %4.1f dB", EbNuwb);
            pamp = sqrt(2.*Tu*65.0/192.0/RateVal/Ts)*pow(10.,(-.05*EbNuwb)); //UWB pulse
amplitude
            npackets=1000;
            for ( i= 0; i< 25; i++) {
                if (i> 10)
                    npackets=2000;
                if (i> 12)
                    npackets=5000;
                if (i> 14)
                    npackets=10000;
                if (i> 16)
                    npackets=20000;
                er=0;
                EbNodb= i* 0.5;
                varn=(65.0/192.0/RateVal/FFTLength)*pow(10.,(-.1*EbNodb)); // calculate
noise variance
                start=clock();
                for ( j= 0; j< npackets; j++) {
                    palen=QpskTx(palen, Bypass,Rate);
                    AwgnChannel(varn, palen, 1);
                    UwbInt2( Tu, Tf, palen, pamp, phi, 1); // Add UWB Interference
                    palen=QpskRx(palen,Bypass,Rate,Decision);
                    er += Error(palen); // Calculate errors in packet
                } //for j
            }
        }
    }

```

```

        //printf( "\nElapsed time=%2.1f seconds\n", duration );
        ber[k][i] = er/(npackets * (palen - 6.0) );
        printf ("\\n # packets=%6.0i EbNo =%4.1f dB, BER = %4.2e", npackets,
EbNodb, ber[k][i]);
            i=i;
        } // for i
    } //for k
} //if
if(!strcmp(Mod, Qam16)){ // 16-Qam modulation
    for ( k= 0; k< 2; k++) {
        EbNuwb=6.0 + 4.*k;
    }
    if (k==0)
        EbNuwb=100000;
    // printf("EbNuwb = %4.1f dB", EbNuwb);
    pamp = sqrt(2.*Tu*65.0/384.0/RateVal/Ts)*pow(10.,(-.05*EbNuwb)); //UWB pulse
amplitude
    if (k==4)
        EbNuwb=10000;
    npackets=1000;
    for ( i= 0; i< 34; i++) {
        if (i> 12)
            npackets=2000;
        if (i> 18)
            npackets=5000;
        if (i> 26)
            npackets=10000;
        if (i> 30)
            npackets=20000;
        er=0;
        EbNodb= i* 0.5;
        varn=(65.0/384.0/RateVal/FFTLength)*pow(10.,(-.1*EbNodb)); // calculate
noise variance
        start=clock();
        for ( j= 0; j< npackets; j++) {
            palen=QAM16Tx(palen, Bypass,Rate);
            AwgnChannel(varn, palen, 1);
            UwbInt2( Tu, Tf, palen, pamp, phi, 1); // Add UWB Interference
            palen=QAM16Rx(palen, Bypass,Rate, Decision);
            er += Error(palen); // Calculate errors in packet
            if
((j==npackets/5)|(j==2*npackets/5)|(j==3*npackets/5)|(j==4*npackets/5)) printf(".");
            i=i;
        } //for j
        ber[k][i] = er/(npackets * (palen - 6.0) );
        printf ("\\n # packets=%6.0i EbNo =%4.1f dB, BER = %4.2e", npackets,
EbNodb, ber[k][i]);
            i=i;
        } // for i
    } //for k
} //if
if(!strcmp(Mod, Qam64)){ //64- Qam Modulation
    for ( k= 0; k< 3; k++) {
        EbNuwb=k*3.0;
        if (k==0)
            EbNuwb=100000;
        printf("EbNuwb = %4.1f dB", EbNuwb);
        pamp = sqrt(2.*Tu*65.0/576.0/RateVal/Ts)*pow(10.,(-.05*EbNuwb)); //UWB pulse
amplitude
        npackets=1000;
        for ( i= 0; i< 37; i++) {
            if (i> 20)
                npackets=2000;
            if (i> 25)
                npackets= 5000;
            if (i> 29)
                npackets= 10000;
            if (i> 31)
                npackets= 20000;
            er=0;
            EbNodb= i* 0.5;

```

```

        varn=(65.0/576.0/RateVal/FFTLength)*pow(10.,(-.1*EbNodb)); // calculate
noise variance
        start=clock();
        for ( j= 0; j< npackets; j++) {
            palen=QAM64Tx(palen, Bypass,Rate);
            AwgnChannel(varn, palen, 1);
            UwbInt2( Tu, Tf, palen, pamp, phi, 1); // Add UWB Interference
            palen=QAM64Rx(palen,Bypass,Rate, Decision);
            er += Error(palen); // Calculate errors in packet
        } //for j
        ber[k][i] = er/(npackets * (palen - 6.0) );
        printf ("\n # packets=%6.0i EbNo =%4.1f dB, BER = %4.2e", npackets,
EbNodb, ber[k][i]);
        i=i;
    } // for i
    printf("\n");
} //for k
} //if

// CI/=npackets;
finish=clock();
duration = (double)(finish - start) / CLOCKS_PER_SEC;

// Write(Mod,Bypass, Rate, Decision, N1, N2, N);
// free (htp);
}

*****
This function compares the transmitted packet in ppacket[] to
the received packet in decod[] and returns the number of errors.
*****/
```

```

int Error(int palen ){
int j,k,er=0;
union {
    unsigned int d16;
    struct{
        unsigned int d0:1;
        }bit;
    }data1;
//mask off the tail bits in the data
ppacket[palen/16-1] &= 0x03ff;
decod [palen/16-1] &= 0x03ff;

for (j = 0; j < palen/16; j++){
    data1.d16 = decod[j]^ppacket[j];
    for (k = 0; k < 16; k++){
        er += data1.bit.d0;
        data1.d16>>=1;
    } //for (k = 0; k < 16; k++)
} //for (j = 0; j < palen; j++)

return (er);
}

*****
This function writes the BER output in a file.
*****/
```

```

void Write( char *str, char *bypass, char *rate, char *Decision, int N1, int N2, int
N){
    FILE *fp;
    char string[80],buf[10];
    int i,j;

    strcpy(string, str);
    if (!strcmp(bypass,"No")){
        if (!strcmp(rate,"1/2"))
            strcat(string,"12");
        if (!strcmp(rate,"2/3"))
            strcat(string,"23");
    }
}
```

```

    if (!strcmp(rate,"3/4"))
        strcat(string,"34");
        strcat(string,Decision);
    }
itoa(N,buf,10);
strcat(string,"N");
strcat(string,buf);
strcat(string,".m");
if( (fp=fopen(string,"wb"))==NULL)
    printf("can't open iqin.dat");
fprintf(fp,"Simulation of BER for %s OFDM in the AWGN\n",str);
if (!strcmp(bypass,"No")){
    fprintf(fp,"Convolutional Code rate=%s, %s Decision\n",rate, Decision);
}
else
    fprintf(fp,"No Convolutional Code \n");

fprintf(fp,"Interference 1/T=%5.2e Hz, N=%2d \n",1/SampleTime, N);
fprintf(fp," \nEbNuwb = ");
for (i=0;i<N1;i++)
    fprintf(fp, " %2d dB ", i*4+8);

fprintf(fp," \nEbNo = 0,1,...%2d dB \n", N2-1);
for (i=0;i<N1; i++){
    fprintf(fp," \nBERBpsk%2d=[ ",4*i+8) ;
    for (j=0; j<N2; j++){
        fprintf(fp,"%4.1e, ",ber[i][j]) ;
    }
    fprintf(fp,"]; ") ;
}
}

/*********************************************
This function calculates the length of the packet according to
the modulation type and coding rate.
******/
int LengthCal(char *Mod,char *Bypass, char *Rate){

    if( !strcmp(Mod, Bpsk) ){
        if( !strcmp(Bypass, "No") ){
            if( !strcmp(Rate, "1/2" ) )
                return (PackBlock);
            if( !strcmp(Rate, "3/4" ) )
                return (3*PackBlock/2);
        }
        return (2*PackBlock);
    }
    if( !strcmp(Mod, Qpsk) ){
        if( !strcmp(Bypass, "No") ){
            if( !strcmp(Rate, "1/2" ) )
                return (2*PackBlock);
            if( !strcmp(Rate, "3/4" ) )
                return (3*PackBlock);
        }
        return (4*PackBlock);
    }
    if( !strcmp(Mod, Qaml6) ){
        if( !strcmp(Bypass, "No") ){
            if( !strcmp(Rate, "1/2" ) )
                return (4*PackBlock);
            if( !strcmp(Rate, "3/4" ) )
                return (6*PackBlock);
        }
        return (8*PackBlock);
    }
    if( !strcmp(Mod, Qam64) ){
        if( !strcmp(Bypass, "No") ){
            if( !strcmp(Rate, "2/3" ) )
                return (8*PackBlock);
            if( !strcmp(Rate, "3/4" ) )
                return (9*PackBlock);
        }
    }
}

```

```

        }
        return (12*PackBlock);
    }
}

```

## File ofdm.cpp

This file contains additional header-type information.

```
#include "ofdm.h"

long      x1=6666,x2=18888,x3=121,x4=178,x5=2140,x6=25000;
int       x10=666,x20=12348,x30=802,x40=18,x50=22140,x60=250;
unsigned int iseed=0x80000;
short packet[20*PackBlock],ppacket[20*PackBlock/16], decod[20*PackBlock/16];
double Inph[Packbuff], Quad[Packbuff];
double InphI[Packbuff], QuadI[Packbuff];
double Inpho[Packbuff], Quado[Packbuff];
int short data[20*PackBlock];double dataf[20*PackBlock];
char Branch[128];
double CI;

char *Bpsk="Bpsk";
char *Qpsk="Qpsk";
char *Qam16="Qam16";
char *Qam64="Qam64";
```

## File Rx.cpp

This file contains the functions and procedures for processing the OFDM baseband samples to produce output data samples.

```
#include "ofdm.h"

//*****************************************************************************
// This function takes the FFT of the input signal located in 'Inpho' and
// 'Quado' arrays and puts it in the 'Inph' and 'Quad' arrays.
//**************************************************************************

void FFT(int palen){
    int i,k;
    // double Io[2*(int)FFTLength];
    double *Io;
    Io = (double*) malloc( 2*(int)FFTLength*sizeof(double) );
    if( Io == NULL )
        printf( "Insufficient memory available\n" );
    for ( k = 0; k < palen/48; k++){
        for ( i = 0; i <(int)FFTLength; i++){
            Io[2*i]=Inpho[TtlLength*k+i+GrdLength];
            Io[2*i+1]=Quado[TtlLength*k+i+GrdLength];
        }
        four1(Io-1,(unsigned int)FFTLength, -1);
        for ( i = 0; i <(int)FFTLength/2; i++){
            Inph[(int)FFTLength*k+i]=Io[2*i+(int)FFTLength];
            Quad[(int)FFTLength*k+i]=Io[2*i+1+(int)FFTLength];
        }
        for ( i = 0; i <(int)FFTLength/2; i++){
            Inph[(int)FFTLength*k+i+(int)FFTLength/2]=Io[2*i];
            Quad[(int)FFTLength*k+i+(int)FFTLength/2]=Io[2*i+1];
        }
        i=i;
    }
    free (Io);
}
```

```

*****
This function demodulates the baseband BPSK input samples in 'Inph' and puts it
in the 'data' array. 'Inix' and 'dix' determine the index number of the
two arrays respectively and are for data alignment purposes.
*****/




void DEM2(int Inix, int dix){

    if (Inph[Inix] <0)
        data[dix]=0;
    else
        data[dix]=1;

}

*****
This function demodulates the baseband QPSK input samples in 'Inph' and
'Quad' and puts it
in the 'data' array. 'Inix' and 'dix' determine the index number of the
two arrays respectively and are for data alignment purposes.
*****/




void DEM4(int Inix, int dix){

    if (Inph[Inix] <0)
        data[dix]=0;
    else
        data[dix]=1;

    if (Quad[Inix] <0)
        data[dix+1]=0;
    else
        data[dix+1]=1;
}

*****
This function demodulates the baseband 16-QAM input samples in 'Inph' and 'Quad'
and puts it
in the 'data' array. 'Inix' and 'dix' determine the index number of the
two arrays respectively and are for data alignment purposes.
*****/




void DEM16(int Inix, int dix){

    if (Inph[Inix] <-2*Norm16QAM){
        data[dix] =0;
        data[dix+1] =0;
    }
    if ( (Inph[Inix] <0*Norm16QAM) && (Inph[Inix] >-2*Norm16QAM) ){
        data[dix] =0;
        data[dix+1] =1;
    }
    if ( (Inph[Inix] >0*Norm16QAM) && (Inph[Inix] <2*Norm16QAM) ){
        data[dix] =1;
        data[dix+1] =1;
    }
    if (Inph[Inix] >2*Norm16QAM){
        data[dix] =1;
        data[dix+1] =0;
    }
    if (Quad[Inix] <-2*Norm16QAM){
        data[dix+2] =0;
        data[dix+3] =0;
    }
    if ( (Quad[Inix] <0*Norm16QAM) && (Quad[Inix] >-2*Norm16QAM) ){
        data[dix+2] =0;
        data[dix+3] =1;
    }
    if ( (Quad[Inix] >0*Norm16QAM) && (Quad[Inix] <2*Norm16QAM) ){
        data[dix+2] =1;
        data[dix+3] =1;
    }
}

```

```

if (Quad[Inix] >2*Norm16QAM){
    data[dix+2] =1;
    data[dix+3] =0;
}
}

***** This function demodulates the baseband 64-Qam input samples in 'Inph' and 'Quad' and
puts it
in the 'data' array. 'Inix' and 'dix' determine the index number of the
two arrays respectively and are for data alignment purposes.
*****


void DEM64(int Inix, int dix){

    if (Inph[Inix] <-6*Norm64QAM){
        data[dix] =0;
        data[dix+1] =0;
        data[dix+2] =0;
    }
    if ( (Inph[Inix] >-6*Norm64QAM) && (Inph[Inix] <-4*Norm64QAM) ){
        data[dix] =0;
        data[dix+1] =0;
        data[dix+2] =1;
    }
    if ( (Inph[Inix] >-4*Norm64QAM) && (Inph[Inix] <-2*Norm64QAM) ){
        data[dix] =0;
        data[dix+1] =1;
        data[dix+2] =1;
    }
    if ( (Inph[Inix] >-2*Norm64QAM) && (Inph[Inix] <0*Norm64QAM) ){
        data[dix] =0;
        data[dix+1] =1;
        data[dix+2] =0;
    }
    if ( (Inph[Inix] >0*Norm64QAM) && (Inph[Inix] <2*Norm64QAM) ){
        data[dix] =1;
        data[dix+1] =1;
        data[dix+2] =0;
    }
    if ( (Inph[Inix] >2*Norm64QAM) && (Inph[Inix] <4*Norm64QAM) ){
        data[dix] =1;
        data[dix+1] =1;
        data[dix+2] =1;
    }
    if ( (Inph[Inix] >4*Norm64QAM) && (Inph[Inix] <6*Norm64QAM) ){
        data[dix] =1;
        data[dix+1] =0;
        data[dix+2] =1;
    }

    if (Inph[Inix] >6*Norm64QAM){
        data[dix] =1;
        data[dix+1] =0;
        data[dix+2] =0;
    }
    if (Quad[Inix] <-6*Norm64QAM){
        data[dix+3] =0;
        data[dix+4] =0;
        data[dix+5] =0;
    }
    if ( (Quad[Inix] >-6*Norm64QAM) && (Quad[Inix] <-4*Norm64QAM) ){
        data[dix+3] =0;
        data[dix+4] =0;
        data[dix+5] =1;
    }
    if ( (Quad[Inix] >-4*Norm64QAM) && (Quad[Inix] <-2*Norm64QAM) ){
        data[dix+3] =0;
        data[dix+4] =1;
        data[dix+5] =1;
    }
    if ( (Quad[Inix] >-2*Norm64QAM) && (Quad[Inix] <-0*Norm64QAM) ){

```

```

        data[dix+3] =0;
        data[dix+4] =1;
        data[dix+5] =0;
    }
    if ( (Quad[Inix] >0*Norm64QAM) && (Quad[Inix] <2*Norm64QAM) ){
        data[dix+3] =1;
        data[dix+4] =1;
        data[dix+5] =0;
    }
    if ( (Quad[Inix] >2*Norm64QAM) && (Quad[Inix] <4*Norm64QAM) ){
        data[dix+3] =1;
        data[dix+4] =1;
        data[dix+5] =1;
    }
    if ( (Quad[Inix] >4*Norm64QAM) && (Quad[Inix] <6*Norm64QAM) ){
        data[dix+3] =1;
        data[dix+4] =0;
        data[dix+5] =1;
    }
    if (Quad[Inix] >6*Norm64QAM){
        data[dix+3] =1;
        data[dix+4] =0;
        data[dix+5] =0;
    }
}

//*****************************************************************************
/* This function demodulates the baseband input array, 'Inph' and puts it
in the 'data' array. The purpose of this function is to find the data sub carriers
which are mixed with pilot and guard carriers and pass it to the DEM2()
***** */
void BPSKDem(int palen){

    int k,i;
    int offset=((int)FFTLength-64)/2;
    for ( k = 0; k < palen/48; k++){
        for (i = 6+offset; i <11+offset; i++)
            DEM2((int)FFTLength*k+i,48*k+i-6-offset);
        for (i = 12+offset; i <25+offset; i++)
            DEM2((int)FFTLength*k+i,48*k+i-7-offset);
        for (i = 26+offset; i <32+offset; i++)
            DEM2((int)FFTLength*k+i,48*k+i-8-offset);
        for (i = 33+offset; i <39+offset; i++)
            DEM2((int)FFTLength*k+i,48*k+i-9-offset);
        for (i = 40+offset; i <53+offset; i++)
            DEM2((int)FFTLength*k+i,48*k+i-10-offset);
        for (i = 54+offset; i <59+offset; i++)
            DEM2((int)FFTLength*k+i,48*k+i-11-offset);
        i=i;
    }
}

//*****************************************************************************
/* This function demodulates the baseband input arrays: 'Inph' and "Quad' and puts it
in the 'data' array. The purpose of this function is to find the data sub carriers
which are mixed with pilot and guard carriers and pass it to the DEM4()
***** */
int QPSKDem(int palen){

    int k,i;
    int offset=((int)FFTLength-64)/2;
    for ( k = 0; k < palen/48; k++){
        for (i = 6+offset; i <11+offset; i++)
            DEM4((int)FFTLength*k+i, 96*k+2*(i-6-offset));
        for (i = 12+offset; i <25+offset; i++)
            DEM4((int)FFTLength*k+i, 96*k+2*(i-7-offset));
        for (i = 26+offset; i <32+offset; i++)
            DEM4((int)FFTLength*k+i, 96*k+2*(i-8-offset));
        for (i = 33+offset; i <39+offset; i++)
            DEM4((int)FFTLength*k+i, 96*k+2*(i-9-offset));

```

```

        for (i = 40+offset; i <53+offset; i++)
            DEM4((int)FFTLength*k+i, 96*k+2*(i-10-offset));
        for (i = 54+offset; i <59+offset; i++)
            DEM4((int)FFTLength*k+i, 96*k+2*(i-11-offset));
    }
    return palen*2;
}

//*****************************************************************************
// This function demodulates the baseband input arrays: 'Inph' and "Quad"
// and puts it
// in the 'data' array. The purpose of this function is to find the data sub carriers
// which are mixed with pilot and guard carriers and pass it to the DEM16()
//*************************************************************************/
int QAM16Dem(int palen){

    int k,i;
    int offset=((int)FFTLength-64)/2;
    for ( k = 0; k < palen/48; k++){
        for (i = 6+offset; i <11+offset; i++)
            DEM16((int)FFTLength*k+i, 192*k+4*(i-6-offset));
        for (i = 12+offset; i <25+offset; i++)
            DEM16((int)FFTLength*k+i, 192*k+4*(i-7-offset));
        for (i = 26+offset; i <32+offset; i++)
            DEM16((int)FFTLength*k+i, 192*k+4*(i-8-offset));
        for (i = 33+offset; i <39+offset; i++)
            DEM16((int)FFTLength*k+i, 192*k+4*(i-9-offset));
        for (i = 40+offset; i <53+offset; i++)
            DEM16((int)FFTLength*k+i, 192*k+4*(i-10-offset));
        for (i = 54+offset; i <59+offset; i++)
            DEM16((int)FFTLength*k+i, 192*k+4*(i-11-offset));
    }
    return palen*4;
}

//*****************************************************************************
// This function demodulates the baseband input arrays: 'Inph' and "Quad"
// and puts it
// in the 'data' array. The purpose of this function is to find the data sub carriers
// which are mixed with pilot and guard carriers and pass it to the DEM64()
//*************************************************************************/
int QAM64Dem(int palen){

    int k,i;
    int offset=((int)FFTLength-64)/2;
    for ( k = 0; k < (palen/48); k++){
        for (i = 6+offset; i <11+offset; i++)
            DEM64((int)FFTLength*k+i, 288*k+6*(i-6-offset));
        for (i = 12+offset; i <25+offset; i++)
            DEM64((int)FFTLength*k+i, 288*k+6*(i-7-offset));
        for (i = 26+offset; i <32+offset; i++)
            DEM64((int)FFTLength*k+i, 288*k+6*(i-8-offset));
        for (i = 33+offset; i <39+offset; i++)
            DEM64((int)FFTLength*k+i, 288*k+6*(i-9-offset));
        for (i = 40+offset; i <53+offset; i++)
            DEM64((int)FFTLength*k+i, 288*k+6*(i-10-offset));
        for (i = 54+offset; i <59+offset; i++)
            DEM64((int)FFTLength*k+i, 288*k+6*(i-11-offset));
    }
    return palen*6;
}

// This function bit maps the decoded data in 'data' to 'decod'
void DataCollect(int palen){
    int k;
    for ( k = 0; k < palen/16 ; k++)
        decod[k] =0;
    for ( k = 0; k < palen; k++)
        decod[k/16] |=( data[k]<<(k%16) );
}

```

```

*****
This function is the baseband BPSK receiver. First it calls FFT() to
transfer the input data samples in 'Inpho' and 'Quado' to frequency domain.
Then it calls BPSKDem() to demodulate the data. 'bypassconv' determines whether
convolutional decoding is to be performed in the receiver. 'rate' shows the
coding rate and 'decision' determine HARD or SOFT decision decoding.
*****
```

```

int BpskRx(int palen, char *bypassconv, char *rate, char *decision){
    static int sw=1;
    if (sw)
        printf("\n BPSK MODULATION\n");
    sw=0;

    FFT(palen);
    if (!strcmp(bypassconv, "No")){
        if (!strcmp(decision, "Hard")){
            BPSKDem(palen);
            if (!strcmp(rate, "1/2"))
                palen=DeConv12(palen, bypassconv);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34(palen, bypassconv);
        }
        else {
            BPSKDemSoft(palen);
            if (!strcmp(rate, "1/2"))
                palen=DeConv12Soft(palen);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34Soft(palen);
        }
    }
    else
        BPSKDem(palen);
    DataCollect(palen);
    return palen;
}

*****
This function is the baseband QPSK receiver. First it calls FFT() to
transfer the input data samples in 'Inpho' and 'Quado' to frequency domain.
Then it calls QPSKDem() to demodulate the data. 'bypassconv' determines whether
convolutional decoding is to be performed in the receiver. 'rate' shows the
coding rate and 'decision' determine HARD or SOFT decision decoding.
*****
```

```

int QpskRx(int palen, char *bypassconv, char *rate, char *decision){
    static int sw=1;
    if (sw)
        printf("\n QPSK MODULATION\n");
    sw=0;

    FFT(palen);
    if (!strcmp(bypassconv, "No")){
        if (!strcmp(decision, "Hard")){
            palen=QPSKDem(palen);
            if (!strcmp(rate, "1/2"))
                palen=DeConv12(palen, bypassconv);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34(palen, bypassconv);
        }
        else {
            palen=QPSKDemSoft(palen);
            if (!strcmp(rate, "1/2"))
                palen=DeConv12Soft(palen);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34Soft(palen);
        }
    }
    else
        palen=QPSKDem(palen);
    DataCollect(palen);
}

```

```

        return palen;
    }

/**************************************************************************
This function is the baseband Qam16 receiver. First it calls FFT() to
transfer the input data samples in 'Inpho' and 'Quado' to frequency domain.
Then it calls QAM16Dem()to demodulate the data. 'bypassconv' determines whether
convolutional decoding is to performed in the receiver. 'rate' shows the
coding rate and 'decision' determine HARD or SOFT decision decoding.
***** */

int QAM16Rx(int palen, char *bypassconv, char *rate, char *decision){
    static int sw=1;
    if (sw)
        printf("\n 16-QAM MODULATION\n");
    sw=0;

    FFT(palen);

    if (!strcmp(bypassconv, "No")){
        if (!strcmp(decision, "Hard")){
            palen=QAM16Dem(palen);
            if (!strcmp(rate, "1/2"))
                palen=DeConv12(palen, bypassconv);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34(palen, bypassconv);
        } //hard
        else {
            palen=QAM16DemSoft(palen);
            if (!strcmp(rate, "1/2"))
                palen=DeConv12Soft(palen);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34Soft(palen);
        } //soft
    } //No
    else{
        palen=QAM16Dem(palen);
    }
    DataCollect(palen);
    return palen;
}

/**************************************************************************
This function is the baseband 64-Qam receiver. First it calls FFT() to
transfer the input data samples in 'Inpho' and 'Quado' to frequency domain.
Then it calls QAM64Dem()to demodulate the data. 'bypassconv' determines whether
convolutional decoding is to performed in the receiver. 'rate' shows the
coding rate and 'decision' determine HARD or SOFT decision decoding.
***** */

int QAM64Rx(int palen, char *bypassconv, char *rate, char *decision){
    static int sw=1;
    if (sw)
        printf("\n 64-QAM MODULATION\n");
    sw=0;

    FFT(palen);
    if (!strcmp(bypassconv, "No")){
        if (!strcmp(decision, "Hard")){
            palen=QAM64Dem(palen);
            if (!strcmp(rate, "2/3"))
                palen=DeConv23(palen, bypassconv);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34(palen, bypassconv);
        }
        else {
            palen=QAM64DemSoft(palen);
            if (!strcmp(rate, "2/3"))
                palen=DeConv23Soft(palen);
            if (!strcmp(rate, "3/4"))
                palen=DeConv34Soft(palen);
        }
    }
}

```

```

        }
    else
        palen=QAM64Dem(palen);
    DataCollect(palen);
    return palen;
}

```

## File RxSoft.cpp

This file contains functions and procedures for implementing soft decisions on the OFDM outputs for soft-decision decoding.

```

#include "ofdm.h"

//*********************************************************************
// This function demodulates the baseband BPSK input samples in 'Inph' and puts it
// in the 'dataf' array. 'Inix' and 'dix' determine the index number of the
// two arrays respectively and are for data alignment purposes. This is soft
// decision demapping
//********************************************************************

void DEM2Soft(int Inix, int dix){
    dataf[dix]=Inph[Inix];
}

//*********************************************************************
// This function demodulates the baseband QPSK input samples in 'Inph' and
// 'Quad' and puts it
// in the 'dataf' array. 'Inix' and 'dix' determine the index number of the
// two arrays respectively and are for data alignment purposes. This
// is soft decision demapping.
//********************************************************************

void DEM4Soft(int Inix, int dix){
    dataf[dix]=Inph[Inix];
    dataf[dix+1]=Quad[Inix];
}

//*********************************************************************
// This function demodulates the baseband 16-Qam input samples in 'Inph' and 'Quad'
// and puts it
// in the 'data' array. 'Inix' and 'dix' determine the index number of the
// two arrays respectively and are for data alignment purposes. This is soft
// decision demapping.
//********************************************************************

void DEM16Soft(int Inix, int dix){

    dataf[dix]=Inph[Inix];
    if (Inph[Inix] >0)
        dataf[dix+1]=2*Norm16QAM-Inph[Inix];
    else
        dataf[dix+1]=Inph[Inix]+2*Norm16QAM;
    dataf[dix+2]=Quad[Inix];
    if (Quad[Inix] >0)
        dataf[dix+3]=2*Norm16QAM-Quad[Inix];
    else
        dataf[dix+3]=Quad[Inix]+2*Norm16QAM;
}

//*********************************************************************
// This function demodulates the baseband 64-Qam input samples in 'Inph' and 'Quad' and
// puts it
// in the 'data' array. 'Inix' and 'dix' determine the index number of the
// two arrays respectively and are for data alignment purposes. This is soft decision
// demapping.
//********************************************************************/

```

```

void DEM64Soft(int Inix, int dix){

    dataf[dix] = Inph[Inix];
    if (Inph[Inix] > 0){
        dataf[dix+1] = 4*Norm64QAM-Inph[Inix];
        if (Inph[Inix] > 4*Norm64QAM)
            dataf[dix+2] = -Inph[Inix]+6*Norm64QAM;
        else
            dataf[dix+2] = -2*Norm64QAM+Inph[Inix];
    }
    if (Inph[Inix] < 0){
        dataf[dix+1] = Inph[Inix]+4*Norm64QAM;
        if (Inph[Inix] < -4*Norm64QAM)
            dataf[dix+2] = Inph[Inix]+6*Norm64QAM;
        else
            dataf[dix+2] = -2*Norm64QAM-Inph[Inix];
    }

    dataf[dix+3] = Quad[Inix];
    if (Quad[Inix] > 0){
        dataf[dix+4] = 4*Norm64QAM-Quad[Inix];
        if (Quad[Inix] > 4*Norm64QAM)
            dataf[dix+5] = -Quad[Inix]+6*Norm64QAM;
        else
            dataf[dix+5] = -2*Norm64QAM+Quad[Inix];
    }
    if (Quad[Inix] < 0){
        dataf[dix+4] = Quad[Inix]+4*Norm64QAM;
        if (Quad[Inix] < -4*Norm64QAM)
            dataf[dix+5] = Quad[Inix]+6*Norm64QAM;
        else
            dataf[dix+5] = -2*Norm64QAM-Quad[Inix];
    }
}

/*********************************************
This function demodulates the baseband input array, 'Inph' and puts it
in the 'data' array. The purpose of this function is to find the data sub carriers
which are mixed with pilot and guard carriers and pass it to the DEM2Soft()
******/

```

```

void BPSKDemSoft(int palen){

    int k,i;
    double cc=0;
    int offset=((int)FFTLength-64)/2;
    for ( k = 0; k < palen/48; k++){
        for (i = 6+offset; i <11+offset; i++)
            DEM2Soft((int)FFTLength*k+i,48*k+i-6-offset);
        for (i = 12+offset; i <25+offset; i++)
            DEM2Soft((int)FFTLength*k+i,48*k+i-7-offset);
        for (i = 26+offset; i <32+offset; i++)
            DEM2Soft((int)FFTLength*k+i,48*k+i-8-offset);
        for (i = 33+offset; i <39+offset; i++)
            DEM2Soft((int)FFTLength*k+i,48*k+i-9-offset);
        for (i = 40+offset; i <53+offset; i++)
            DEM2Soft((int)FFTLength*k+i,48*k+i-10-offset);
        for (i = 54+offset; i <59+offset; i++)
            DEM2Soft((int)FFTLength*k+i,48*k+i-11-offset);
        i=i;
    }
}

/*********************************************
This function demodulates the baseband input arrays: 'Inph' and "Quad"
and puts it
in the 'data' array. The purpose of this function is to find the data sub carriers
which are mixed with pilot and guard carriers and pass it to the DEM4Soft()
******/

```

```

int QPSKDemSoft(int palen){

```

```

int k,i;
int offset=((int)FFTLength-64)/2;

for ( k = 0; k < palen/48; k++){
    for (i = 6+offset; i <11+offset; i++)
        DEM4Soft((int)FFTLength*k+i, 96*k+2*(i-6-offset));
    for (i = 12+offset; i <25+offset; i++)
        DEM4Soft((int)FFTLength*k+i, 96*k+2*(i-7-offset));
    for (i = 26+offset; i <32+offset; i++)
        DEM4Soft((int)FFTLength*k+i, 96*k+2*(i-8-offset));
    for (i = 33+offset; i <39+offset; i++)
        DEM4Soft((int)FFTLength*k+i, 96*k+2*(i-9-offset));
    for (i = 40+offset; i <53+offset; i++)
        DEM4Soft((int)FFTLength*k+i, 96*k+2*(i-10-offset));
    for (i = 54+offset; i <59+offset; i++)
        DEM4Soft((int)FFTLength*k+i, 96*k+2*(i-11-offset));
}
return palen*2;
}

/*********************************************
This function demodulates the baseband input arrays: 'Inph' and "Quad"
and puts it
in the 'data' array. The purpose of this function is to find the data sub carriers
which are mixed with pilot and guard carriers and pass it to the DEM16Soft()
*****
```

int QAM16DemSoft(int palen){

```

int k,i;
int offset=((int)FFTLength-64)/2;
for ( k = 0; k < palen/48; k++){
    for (i = 6+offset; i <11+offset; i++)
        DEM16Soft((int)FFTLength*k+i, 192*k+4*(i-6-offset));
    for (i = 12+offset; i <25+offset; i++)
        DEM16Soft((int)FFTLength*k+i, 192*k+4*(i-7-offset));
    for (i = 26+offset; i <32+offset; i++)
        DEM16Soft((int)FFTLength*k+i, 192*k+4*(i-8-offset));
    for (i = 33+offset; i <39+offset; i++)
        DEM16Soft((int)FFTLength*k+i, 192*k+4*(i-9-offset));
    for (i = 40+offset; i <53+offset; i++)
        DEM16Soft((int)FFTLength*k+i, 192*k+4*(i-10-offset));
    for (i = 54+offset; i <59+offset; i++)
        DEM16Soft((int)FFTLength*k+i, 192*k+4*(i-11-offset));
}
return palen*4;
}

/*********************************************
This function demodulates the baseband input arrays: 'Inph' and "Quad"
and puts it
in the 'data' array. The purpose of this function is to find the data sub carriers
which are mixed with pilot and guard carriers and pass it to the DEM64Soft()
*****
```

int QAM64DemSoft(int palen){

```

int k,i;
double cc=0;
int offset=((int)FFTLength-64)/2;
for ( k = 0; k < palen/48; k++){
    for (i = 6+offset; i <11+offset; i++)
        DEM64Soft((int)FFTLength*k+i, 288*k+6*(i-6-offset));
    for (i = 12+offset; i <25+offset; i++)
        DEM64Soft((int)FFTLength*k+i, 288*k+6*(i-7-offset));
    for (i = 26+offset; i <32+offset; i++)
        DEM64Soft((int)FFTLength*k+i, 288*k+6*(i-8-offset));
    for (i = 33+offset; i <39+offset; i++)
        DEM64Soft((int)FFTLength*k+i, 288*k+6*(i-9-offset));
    for (i = 40+offset; i <53+offset; i++)
        DEM64Soft((int)FFTLength*k+i, 288*k+6*(i-10-offset));
    for (i = 54+offset; i <59+offset; i++)
        DEM64Soft((int)FFTLength*k+i, 288*k+6*(i-11-offset));
}
```

```
        DEM64Soft((int)FFTLength*k+i, 288*k+6*(i-11-offset));
    }
    return palen*6;
}
```

## File Tx.cpp

This file contains the functions and procedures for generating the random data and forming the OFDM symbol.

```

istep=mmax << 1;
theta=isign*(2*PI/mmax);
wtemp=sin(.5*theta);
wpr = -2.0*wtemp*wtemp;
wpi=sin(theta);
wr=1.0;
wi=0.0;
for (m=1; m<mmax;m+=2) {
    for(i=m;i<=n;i+=istep){
        j=i+mmax;
        temp=wr*ddata[j]-wi*ddata[j+1];
        tempi=wr*ddata[j+1]+wi*ddata[j];
        ddata[j]=ddata[i]-temp;
        ddata[j+1]=ddata[i+1]-tempi;
        ddata[i] += temp;
        ddata[i+1] += tempi;
    }
    wr=(wtemp=wr)*wpr-wi*wpi+wr;
    wi=wi*wpr+wtemp*wpi+wi;
}
mmax=istep;
}

int Conv12(int palen){ // Convolutional encoder using rate 1/2
int reg[6]={0};
int i;
int Apacket[Packbuff], Bpacket[Packbuff];

for (i=0; i < palen ; i++){
    Apacket[i]=reg[1]^reg[2]^reg[4]^reg[5]^packet[i];
    Bpacket[i]=reg[0]^reg[1]^reg[2]^reg[5]^packet[i];
    reg[5]=reg[4];
    reg[4]=reg[3];
    reg[3]=reg[2];
    reg[2]=reg[1];
    reg[1]=reg[0];
    reg[0]=packet[i];
}
for (i=0; i < palen ; i++){
    packet[2*i]=Apacket[i];
    packet[2*i+1]=Bpacket[i];
}
return palen*2;
}

int Conv34(int palen){ // convolutional encoder using rate 3/4
int reg[6]={0};
int i;
int Apacket[Packbuff], Bpacket[Packbuff];
static int sw=1;

if (sw)
    printf("\n 3/4 rate convolutional code");
sw=0;
for (i=0; i < palen ; i++){
    Apacket[i]=reg[1]^reg[2]^reg[4]^reg[5]^packet[i];
    Bpacket[i]=reg[0]^reg[1]^reg[2]^reg[5]^packet[i];
    reg[5]=reg[4];
    reg[4]=reg[3];
    reg[3]=reg[2];
    reg[2]=reg[1];
    reg[1]=reg[0];
    reg[0]=packet[i];
}
for (i=0; i < palen/3 ; i++){
    packet[4*i]=Apacket[ 3*i];
    packet[4*i+1]=Bpacket[3*i];

    packet[4*i+2]=Apacket[3*i+1];
    packet[4*i+3]=Bpacket[3*i+2];
}

```

```

    return palen*4/3;

}

int Conv23(int palen){ // convolutional coder using rate 2/3
    int reg[6]={0};
    int i;
    int Apacket[Packbuff], Bpacket[Packbuff];
static int sw=1;

    if (sw)
        printf("\n 2/3 rate convolutional code");
    sw=0;
    for (i=0; i < palen ; i++){
        Apacket[i]=reg[1]^reg[2]^reg[4]^reg[5]^packet[i];
        Bpacket[i]=reg[0]^reg[1]^reg[2]^reg[5]^packet[i];
        reg[5]=reg[4];
        reg[4]=reg[3];
        reg[3]=reg[2];
        reg[2]=reg[1];
        reg[1]=reg[0];
        reg[0]=packet[i];
    }
    for (i=0; i < palen/2 ; i++){
        packet[3*i]=Apacket[ 2*i];
        packet[3*i+1]=Bpacket[2*i];
        packet[3*i+2]=Apacket[2*i+1];
    }
    return palen*3/2;
}

void trellis(void){ // generating trellis output

    int i,k,j;
    char reg=0;
    char dd;
    for (i=0; i< 128; i+=4){
        for (k=0; k< 2; k++){
            dd=k;
            for (j=0; j< 2; j++){
                reg=i/2+j; //states 0,1,2,3,4,....,63 as a binary number (reverse of
register state)
/*B*/
                Branch[i+2*k+j] =
(dd^(reg&0x1)^((reg>>3)&0x1)^((reg>>4)&0x1)^((reg>>5)&0x1) );
                Branch[i+2*k+j] <<=1;
/*A*/
                Branch[i+2*k+j] |=
(dd^(reg&0x1)^((reg>>1)&0x1)^((reg>>3)&0x1)^((reg>>4)&0x1));
                k=k;
            }
        }
        k=k;
    }
}

void BPSK(int palen){ // Bpsk modulator
    for (int j = 0; j < palen/48; j++){
        for (int i = 0; i <48; i++){
            Inph[48*j+i]=2*packet[48*j+i]-1;
            Quad[48*j+i]=0;
        }
    }
}

int QPSK(int palen){ // Qpsk modulator
    for (int j = 0; j < palen/96; j++){
        for (int i = 0; i <48; i++){
            Inph[48*j+i]=NormQPSK*(2*packet[96*j+2*i]-1);
            Quad[48*j+i]=NormQPSK*(2*packet[96*j+2*i+1]-1);
        }
    }
    return palen/2;
}

int QAM16(int palen){ // 16-Qam modulator

```

```

short table[4]={-3,-1, 3, 1};
for (int j = 0; j < palen/192; j++){
    for (int i = 0; i <48; i++){
        Inph[48*j+i]=Norm16QAM*(table[2*packet[192*j+4*i]+packet[192*j+4*i+1] ] );
        Quad[48*j+i]=Norm16QAM*(table[2*packet[192*j+4*i+2]+packet[192*j+4*i+3] ] );
    }
}
return palen/4;
}

int QAM64(int palen){ //64-Qam modulator
short table[8]={-7, -5, -1, -3, 7, 5, 1, 3};

for (int j = 0; j < palen/288; j++){
    for (int i = 0; i <48; i++){
        Inph[48*j+i]=Norm64QAM*( table
[4*packet[288*j+6*i]+2*packet[288*j+6*i+1]+packet[288*j+6*i+2]] ;
        Quad[48*j+i]=Norm64QAM*( table
[4*packet[288*j+6*i+3]+2*packet[288*j+6*i+4]+packet[288*j+6*i+5]] ;
    }
}
return palen/6;
}

// This function adds the pilot subcarriers to the data subcarriers
// and then takes IFFT by calling fourl().
void IFFT(int palen){
    int i,k;
    double *Io;

Io = (double*) malloc( 2*(int)FFTLength*sizeof(double) );
if( Io == NULL )
    printf( "Insufficient memory available\n" );
for ( k = 0; k < palen/48; k++){
    for (i = 0; i <2*(int)FFTLength; i++)
        Io[i]=0;
//////Positive Freq.
    for (i = 0; i <6; i++){
        Io[2*i+2]=Inph[48*k+i+24];
        Io[(2*i+1)+2]=Quad[48*k+i+24];
    }
    Io[2*6+2]=p127[k%126]*P[2];
    Io[2*6+1+2]=0;
    for (i = 6; i <19; i++){
        Io[2*i+4]=Inph[48*k+i+24];
        Io[(2*i+1)+4]=Quad[48*k+i+24];
    }
    Io[2*20+2]=p127[k%126]*P[3];
    Io[2*20+1+2]=0;
    for (i = 19; i <24; i++){
        Io[2*i+6]=Inph[48*k+i+24];
        Io[2*i+1+6]=Quad[48*k+i+24];
    }
///////////Negative Frequency
    for (i = 0; i <6; i++){
        Io[2*(int)FFTLength-(2*i+1)-1]=Inph[48*k+23-i];
        Io[2*(int)FFTLength-2*i-1]=Quad[48*k+23-i];
    }
    i=6;
    Io[2*(int)FFTLength-(2*i+1)-1]=p127[k%126]*P[1];
    Io[2*(int)FFTLength-2*i-1]=0;
    for (i = 6; i <19; i++){
        Io[2*(int)FFTLength-(2*i+1)-3]=Inph[48*k+23-i];
        Io[2*(int)FFTLength-2*i-3]=Quad[48*k+23-i];
    }
    i=19;
    Io[2*(int)FFTLength-(2*i+1)-3]=p127[k%126]*P[0];
    Io[2*(int)FFTLength-2*i-3]=0;
    for (i = 19; i <24; i++){
        Io[2*(int)FFTLength-(2*i+1)-5]=Inph[48*k+23-i];
        Io[2*(int)FFTLength-2*i-5]=Quad[48*k+23-i];
    }
}

```

```

        }
        fourl(Io-1,(unsigned int)FFTLength, 1);
        for (i = 0; i <FFTLength; i++){
            Inpho[TtlLength*k+i+GrdLength]=Io[2*i]/FFTLength;
            Quado[TtlLength*k+i+GrdLength]=Io[2*i+1]/FFTLength;
        }
        i=i;
    }
    free( Io );
}

// This function adds a guard time to the fft sample by cyclically
// extending the signal.
void AddGuard(int palen){
    int i,j;

    for ( j = 0; j < palen/48; j++){
        for (i=0;i<GrdLength;i++){
            Inpho[TtlLength*j+(GrdLength-1)-i]=Inpho[TtlLength*j+TtlLength-2-i];
            Quado[TtlLength*j+(GrdLength-1)-i]=Quado[TtlLength*j+TtlLength-2-i];
        }
        Inpho[TtlLength*j+TtlLength-1]=Inpho[TtlLength*j+GrdLength];
        Quado[TtlLength*j+TtlLength-1]=Quado[TtlLength*j+GrdLength];
        i=i;
    }
}

/*********************************************
This function generates a random packet of data by calling DataGen(),
it encodes the input data if the bypassconv parameter is set to "NO, it
modulates the signal using BPSK by calling BPSK(), it takes the ifft and
then it adds a guard time to the signal.
********************************************/

int BpskTx(int palen, char *bypassconv, char *rate){
    DataGen(palen);
    if (!strcmp(bypassconv,"No")){
        if (!strcmp(rate,"1/2"))
            palen=Conv12(palen);
        if (!strcmp(rate,"3/4"))
            palen=Conv34(palen);
    }
    BPSK(palen);
    IFFT(palen);
    AddGuard(palen);
    return palen;
}

/*********************************************
This function generates a random packet of data by calling DataGen(),
it encodes the input data if the bypassconv parameter is set to "NO, it
modulates the signal using QPSK by calling QPSK(), it takes the ifft and
then it adds a guard time to the signal.
********************************************/

int QpskTx(int palen, char *bypassconv, char *rate){
    DataGen(palen);
    if (!strcmp(bypassconv,"No")){
        if (!strcmp(rate,"1/2"))
            palen=Conv12(palen);
        if (!strcmp(rate,"3/4"))
            palen=Conv34(palen);
    }
    palen=QPSK(palen);
    IFFT(palen);
    AddGuard(palen);
    return palen;
}

/*********************************************
This function generates a random packet of data by calling DataGen(),
it encodes the input data if the bypassconv parameter is set to "NO, it

```

```

modulates the signal using 16-Qam by calling QAM16(), it takes the ifft and
then it adds a guard time to the signal.
***** */

int QAM16Tx(int palen, char *bypassconv, char *rate){
    DataGen(palen);
    if (!strcmp(bypassconv,"No")){
        if (!strcmp(rate,"1/2"))
            palen=Conv12(palen);
        if (!strcmp(rate,"3/4"))
            palen=Conv34(palen);
    }
    palen=QAM16(palen);
    IFFT(palen);
    AddGuard(palen);
    return palen;
}

***** 
This function generates a random packet of data by calling DataGen(),
it encodes the input data if the bypassconv parameter is set to "NO", it
modulates the signal using 64-QAM by calling QAM64(), it takes the ifft and
then it adds a guard time to the signal.
***** */

int QAM64Tx(int palen, char *bypassconv, char *rate){
    DataGen(palen);
    if (!strcmp(bypassconv,"No")){
        if (!strcmp(rate,"2/3"))
            palen=Conv23(palen);
        if (!strcmp(rate,"3/4"))
            palen=Conv34(palen);
    }
    palen=QAM64(palen);
    IFFT(palen);
    AddGuard(palen);
    return palen;
}

```

## REFERENCES

- [1] B. P. Lathi, *Linear Systems and Signals*. Carmichael, California: Berkeley-Cambridge Press, 1992.
- [2] ——, IEEE Standard 802.11a-1999, “Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” 1999.
- [3] L. E. Miller, “Models Of UWB Waveforms and Their Effects on Narrowband Direct Conversion Receivers,” accepted for presentation at UWBST 2003, Nov. 16-18, 2003.
- [4] J. G. Proakis, *Digital Communications* (4th edition). New York: McGraw-Hill, 2001.
- [5] J. M. Wozencraft and I. M. Jacobs, *Principles of Communications Engineering*. New York: Wiley, 1965.
- [6] I. S. Gradshteyn and I. M. Ryzhik, *Table of Integrals, Series and Products*. New York: Academic Press, 1965.
- [7] W. Gautschi and W. F. Cahill, “Exponential Integral and Related Functions,” in *Handbook of Mathematical Functions*, National Bureau of Standards [now NIST] Applied Mathematics Series # 55. Washington: Government Printing Office, 1970.
- [8] J. S. Lee and L. E. Miller, *CDMA Systems Engineering Handbook*. Boston: Artech House, 1998.
- [9] M. Z. Win, “Spectral Density of Random UWB Signals,” *IEEE Communications Letters*, Vol. 6, pp. 526-528 (December 2002).
- [10] J. Conan, “The Weight Spectra of Some Short Low-Rate Convolutional Codes,” *IEEE Trans. On Commun. (Correspondence)*, Vol. COM-32, pp. 1050-1053 (September 1984).
- [11] H. H. Ma and M. A. Poole, “Error-Correcting Codes Against the Worst-Case Partial-Band Jammer,” *IEEE Trans. On Commun.*, Vol. COM-32, pp. 124-133 (February 1984).
- [12] J. P. Odenwalder, “Optimal Decoding of Convolutional Codes,” UCLA Ph.D. dissertation, 1970, pp. 62-68.
- [13] D. Haccoun and G. Begin, “High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding,” *IEEE Trans. On Commun.*, Vol. 37, pp. 1113-1125 (November 1989).
- [14] R. van Nee and R. Prasad, *OFDM for Wireless Multimedia Applications*. Boston: Artech House, 2000.
- [15] P. M. Woodward, *Probability and Information Theory, with Applications to Radar* (2nd ed.). New York: Pergamon Press, 1964. Republished in 1980 by Artech House.
- [16] M. Welborn and K. Siwiak, eds., “Ultra-Wideband Tutorial,” IEEE 802.15 document 02/133r1. Available online at  
[http://grouper.ieee.org/groups/802/15/pub/2002/Mar02/02133r1P802-15\\_WG-Ultra-Wideband-Tutorial.ppt](http://grouper.ieee.org/groups/802/15/pub/2002/Mar02/02133r1P802-15_WG-Ultra-Wideband-Tutorial.ppt).
- [17] A. Soltanian, “Coexistence of Ultra-Wideband System and IEEE 802.11a WLAN,” NIST WCTG report, April 2003 (Revised October 2003).