mobile communications series

SOFTWARE-DEFINED RADIO for ENGINEERS

TRAVIS F. COLLINS
ROBIN GETZ
DI PU
ALEXANDER M. WYGLINSKI

Software-Defined Radio for Engineers

For a listing of recent titles in the *Artech House Mobile Communications*, turn to the back of this book.

Software-Defined Radio for Engineers

Travis F. Collins
Robin Getz
Di Pu
Alexander M. Wyglinski

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalog record for this book is available from the British Library.

ISBN-13: 978-1-63081-457-1

Cover design by John Gomes

10 9 8 7 6 5 4 3 2 1

Dedication

To my wife Lauren
—Travis Collins

To my wonderful children, Matthew, Lauren, and Isaac, and my patient wife, Michelle—sorry I have been hiding in the basement working on this book. To all my fantastic colleagues at Analog Devices: Dave, Michael, Lars-Peter, Andrei, Mihai, Travis, Wyatt and many more, without whom Pluto SDR and IIO would not exist.
—Robin Getz

To my lovely son Aidi, my husband Di, and my parents Lingzhen and Xuexun
—Di Pu

To my wife Jen
—Alexander Wyglinski

Contents

CHAPTER 8

Frame Synchronization and Channel Coding

In this chapter we will cover the topics of frame synchronization and channel coding. First, frame synchronization will be discussed to complete our full reception of data from the transmitter. As in Chapter 7, which required timing recovery to realize, for frame synchronization to be accomplished, it requires that the signal has been timing and frequency corrected. However, once frame synchronization has been completed we can fully decode data over our wireless link. Once this has been accomplished, we can move on toward channel coding, where we will discuss popular coding techniques and some of the implementation details from the perspective of a system integrator.

With regard to our receiver outline in Figure 8.1, this chapter will address the second-to-last block, Frame Sync, which is highlighted.

8.1 O Frame, Where Art Thou?

In previous chapters we have discussed frequency correction, timing compensation, and matched filtering. The final aspect of synchronization is frame synchronization. At this point it is assumed that the available samples represent single symbols and are corrected for timing, frequency, and phase offsets. However, since in a realistic system the start of a frame will still be unknown, we need to perform an additional correction or estimation. We demonstrate this issue visually in Figure 8.2, which contains a sample synchronized frame with an unknown offset of p samples. Mathematically, this is simply an unknown delay in our signal y:

$$u[n] = y[n - p], \tag{8.1}$$

where $p \in \mathbb{Z}$. Once we have an estimate $\hat{p}$ we can extract data from the desired frame, demodulated to bits, and perform any additional channel decoding or source decode originally applied to the signal. There are various way to accomplish this estimation but the implemented outline in this chapter is based around cross-correlation.

Depending on the receiver structure and waveform it may be possible to perform frame synchronization after demodulation, where we mark the start of a frame with a specific sequence of bits. However, this cannot be used if symbols are required downstream for an equalizer or if the preamble contains configuration parameters for downstream modulation. This is the case in IEEE 802.11 [1], where the preamble can have a different modulation than the payload. Alternatively, if the system is packet-based and does not continuously transmit data it can be difficult

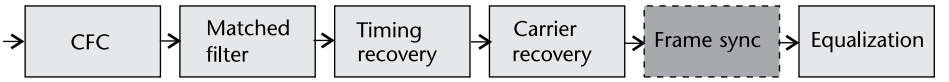


Figure 8.1 Receiver block diagram.

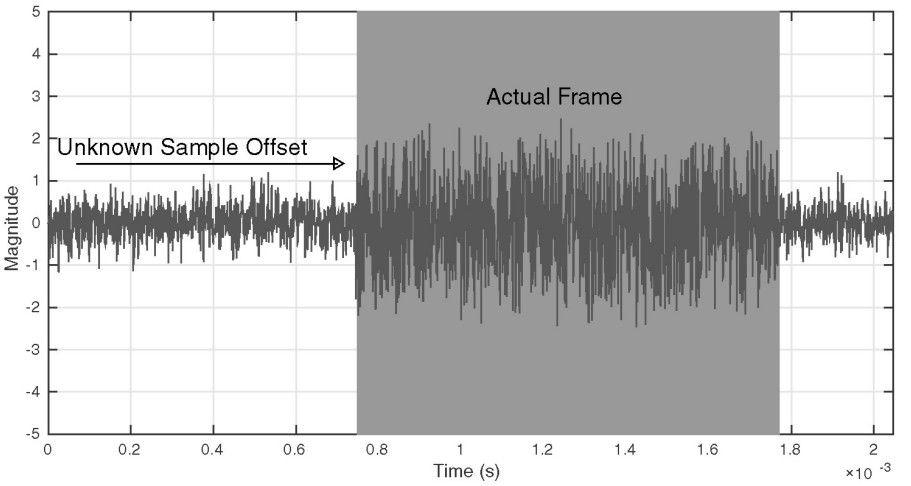


Figure 8.2 Example received frame in AWGN with an unknown sample offset.

to distinguish noise from actual received signal. However, if only bits are consult the relative gain of the signal is removed, which is information that is useful when determining if signal is present.

8.2 Frame Synchronization

The common method of determining the start of a given frame is with the use of markers, even in wired networking. However, in the case of wireless signals, this problem becomes more difficult, as made visible in Figure 8.2, which actually uses a marker. Due to the high degree of noise content in the signal, specifically designed preamble sequences are appended to frames before modulation. Such sequences are typically known exactly at the receiver and have certain qualities that make frame estimation accurate. In Figure 8.3 we outline a typical frame ordering containing preamble, header, and payload data. Header and payloads are unknown are the receiver, but will maintain some structure so they can be decoded correctly.

Before we discuss the typical sequences utilized we will introduce a technique for estimation of the start of a known sequence starting at an unknown sample in time. Let us consider a set of N different binary sequences b_n, where $n \in [1, ..., N]$, each of length L. Given an additional binary sequence d, we want to determine how similar d is to the existing N sequences. The use of a cross correlation would provide us the appropriate estimate, which we perform as

$$C_{d,b}(k) = \sum_{m} d^*(m) b_n(m+k), \tag{8.2}$$

which is identical to a convolution without a time reversal on the second term. When $d = b_n$ for a given n, $C_{d,b}$ will be maximized compared with the other $n-1$

Figure 8.3 Common frame structure of wireless packet with preamble followed by header and payload data.

sequences, and produce a peak at L^{th} index at least. We can use this concept to help build our frame start estimator, which as discussed will contain a known sequence called the preamble.

Common sequences utilized in preambles for narrowband communications are Barker codes [2]. Barker codes are utilized since they have unique autocorrelation properties that have minimal or ideal off-peak correlation. Specifically, such codes or sequences $a(i)$ have autocorrelation functions defined as

$$c(k) = \sum_{i=1}^{N-k} a(i)a(i+k), \tag{8.3}$$

such that

$$|c(v)| \leq 1, \quad 1 \leq v < N. \tag{8.4}$$

However, only nine sequences are known $N \in [1, 2, 3, 4, 5, 7, 11, 13]$, provided in Table 8.1. We provide a visualization of these autocorrelations in Figure 8.5 for a select set of lengths. As the sequence becomes longer the central peak becomes more pronounced. For communication systems we typically append multiple such codes together to produce longer sequences for better performance, as well as for other identifications.

Using these codes we have implemented a small example to show how a Barker sequence $a(k)$ can be used to locate sequences in a larger set of data $r(k)$, which we have provided in Code 8.1. In this example we insert a Barker code within a larger random sequence at an unknown position p, similar to our original error model in Section 8.1, shown as Figure 8.4(a). A cross correlation is performed using MATLAB's `xcorr` function, provided in Figure 8.4(b). The cross correlation will be of length $2L_r - 1$, where L_r is the length of r. Since `xcorr` will pad zeros to a so its length is equal to L_r [3], this will result in at least $L_r - L_a$ zeros to appear in the correlation where L_a is the original length of a. From Figure 8.5, we know that the peak will appear at L_a samples from the start of the sequence. Taking this into account we can directly determine at what offset position of our desired sequence:

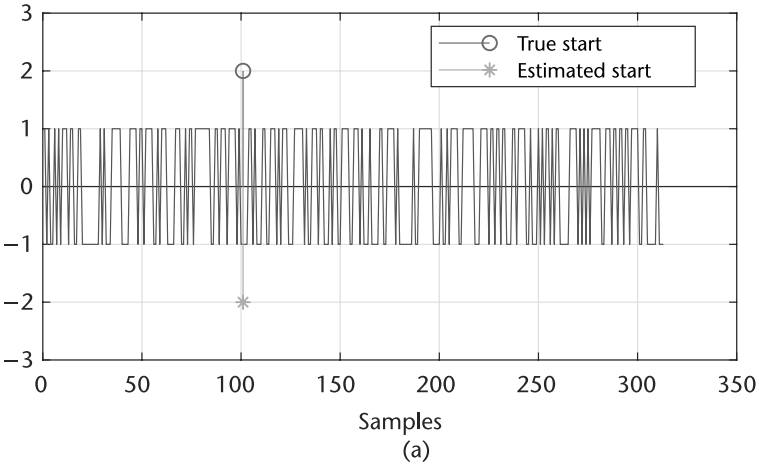
$$\hat{p} = \underset{k}{\text{argmax}}\ C_{ra}(k) - L_r, \tag{8.5}$$

which is what we observe from our estimates in Figure 8.4(a).

The `xcorr` function is a useful tool in MATLAB and will actually utilize the `fft` function for large sequences for performance. Since we know from Chapter 2 that convolution is just a multiplication in the frequency domain, and from above the relation of correlation and convolution, this strategy is obvious for `xcorr`. However, the process actually inflates the data processed since the sequences must be of equal length for correlation. We can observe inflation from the zeros in

Table 8.1 Barker Codes from `comm.BarkerCode`

N	*Code*
2	−1, +1
3	−1, −1, +1
4	−1, −1, +1, −1
5	−1, −1, −1, +1, −1
7	−1, −1, −1, +1, +1, −1, +1
11	−1, −1, −1, +1, +1, +1, −1, +1, +1, −1, +1
13	−1, −1, −1, −1, −1, +1, +1, −1, −1, +1, −1, +1, −1



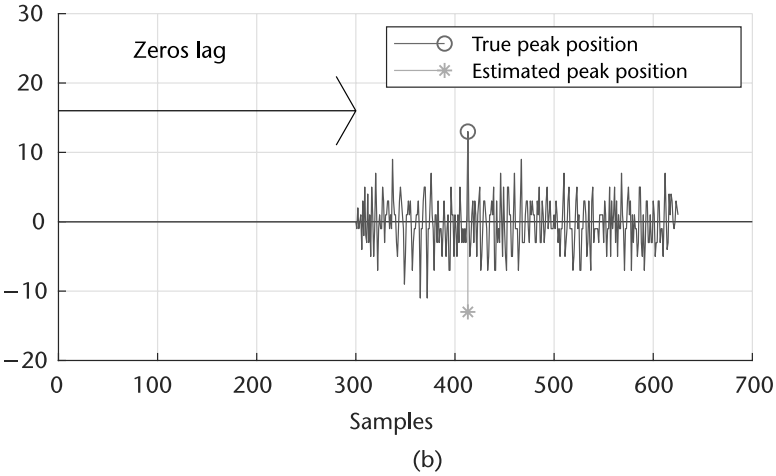


Figure 8.4 Example of using cross correlation to find a sequence with a larger sequence of data. (a) Random bit sequence with Barker code embedded at delay p, and (b) crosscorrelation of Barker code with random sequence containing code.

Figure 8.4(b). A more efficient implementation would be to utilize a filter. The output y of an FIR filter with taps b_i can be written as

$$y[n] = \sum_{i=0}^{N} b_i \, u[n-i], \tag{8.6}$$

where u is our received signal that contains the sequence of interest. Equation (8.6) is almost identical to (8.2) except for a time reversal. Therefore, to use an FIR filter

Code 8.1 Barker Sequence Example: **barkerBits13.m**

```
% Show Barker Autocorrelations search example
sequenceLength = 13;
hBCode = comm.BarkerCode('Length',7,'SamplesPerFrame', sequenceLength);
seq = hBCode(); gapLen = 100; gapLenEnd = 200;
gen = @(Len) 2*randi([0 1],Len,1)-1;
y = [gen(gapLen); seq; gen(gapLenEnd)];
corr = xcorr(y,seq);
L = length(corr);
[v,i] = max(corr);
% Estimation of peak position
% The correlation sequence should be 2*L-1, where L is the length of the
% longest of the two sequences
%
% The first N-M will be zeros, where N is the length of the long sequence
% and N is the length of the shorter sequence
%
% The peak itself will occur at zero lag, or when they are directly
```

as a cross correlator we could simply replace b_i with the sequence of interest, but in reverse order. This implementation would not require padding of the sequence of interest d, and can be efficiently implemented in hardware.

Based on Code 8.1, reimplement the sequence search with an FIR filter.

8.2.1 Signal Detection

Now that we have a method for estimating the start of a frame, let us consider a slightly simpler problem. Can we determine that a frame exists in the correlation? This can be useful if wish to handle data in smaller pieces rather than working with complete frames, or possibly a mechanism for determining if a channel is occupied. When we consider signal detection, we typically define this feature as a power sensitivity or the minimum received power at the receiver to be detected. However, this sensitivity will be based on some source waveform and cannot be generalized in most cases. Therefore, such a value should never be given on its own, unless given with respect to some standard transmission. Even when considering formal methods of detection theory, such as Neyman-Pearson or even Bayesian, you must have some knowledge or reference to the source signal [4]. The receiver sensitivity requirement for IEEE 802.11ac specifically is defined as the minimum received signal power to maintain a packet error rate of 10%, for a give modulation and coding scheme [1].

In a formal mathematic context, this process devolves into a simple binary hypothesis test:

$$\begin{aligned} \mathcal{H}_0 &: \text{no signals}, \\ \mathcal{H}_1 &: \text{signals exist}, \end{aligned} \tag{8.7}$$

where $\mathcal{H}_0$ is usually referred to as a null hypothesis and $\mathcal{H}_1$ is usually called an alternative hypothesis.

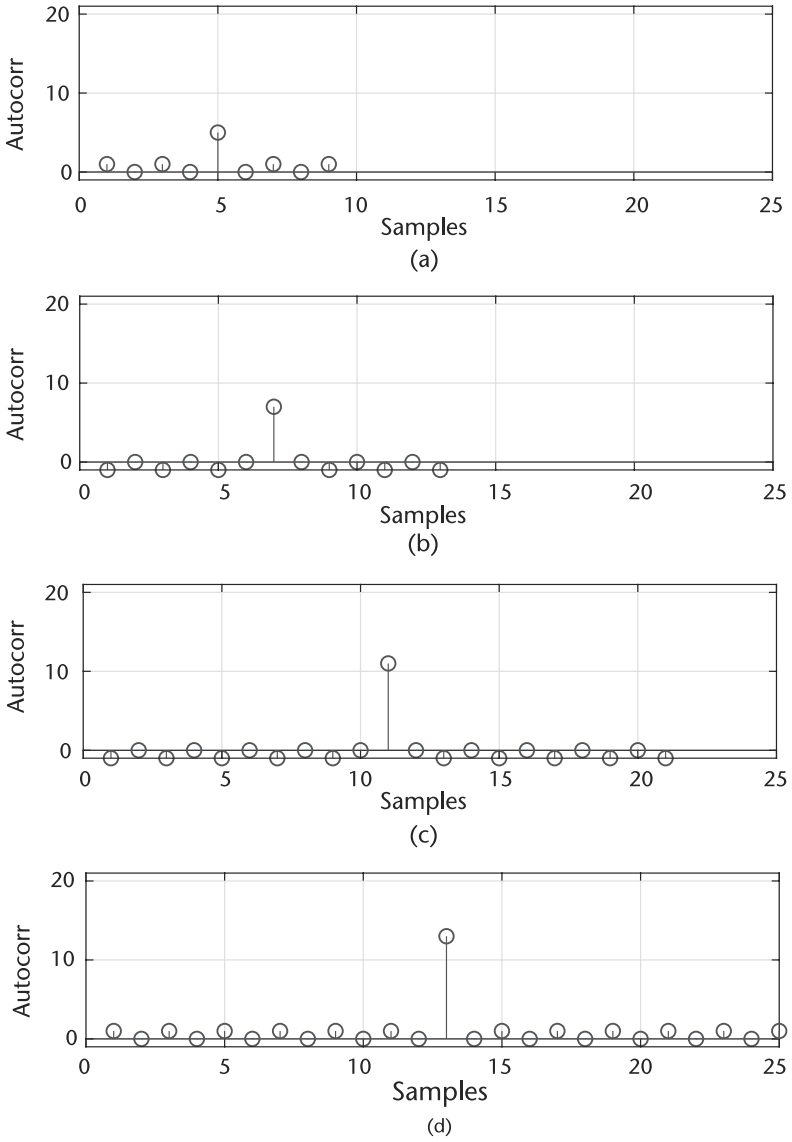


Figure 8.5 Comparison of autocorrelations of Barker sequences of different lengths. (a) $N = 5$, (b) $N = 7$, (c) $N = 11$, and (d) $N = 13$.

For a null hypothesis, since there are no primary signals present, the received signal is just the noise in the RF environment. On the other hand, for the alternative hypothesis, the received signal would be the superposition of the noise and the primary signals. Thus, the two hypotheses in (8.14) can be represented as

$$\begin{aligned} \mathcal{H}_0 &: r[n] = n[n], \\ \mathcal{H}_1 &: r[n] = x[n] + n[n], \end{aligned} \tag{8.8}$$

where $r[n]$ is the received signal, $n[k]$ is the noise in the RF environment, and $x[n]$ is the signal we are trying to detect. Based on the observation r, we need to decide among two possible statistical situations describing the observation, which can be

expressed as

$$\delta(x) = \begin{cases} 1 & x \in \Gamma_1, \\ 0 & x \in \Gamma_1^c. \end{cases} \tag{8.9}$$

When the observation x falls inside the region Γ_1, we will choose $\mathcal{H}_1$. However, if the observation falls outside the region Γ_1, we will choose $\mathcal{H}_0$. Therefore, (8.9) is known as *decision rule*, which is a function that maps an observation to an appropriate hypothesis [5]. In the context of packet detection, thresholding is actually the implementation of a decision rule.

Regardless of the precise signal model or detector used, sensing errors are inevitable due to additive noise, limited observations, and the inherent randomness of the observed data [6]. In testing $\mathcal{H}_0$ versus $\mathcal{H}_1$ in (8.14), there are two types of errors that can be made; namely, $\mathcal{H}_0$ can be falsely rejected or $\mathcal{H}_1$ can be falsely rejected [5]. In the first hypothesis, there are actually no signals in the channel, but the testing detects an occupied channel, so this type of error is called a *false alarm* or *Type I error*. In the second hypothesis, there actually exist signals in the channel, but the testing detects only a vacant channel. Thus, we refer to this type of error as a *missed detection* or *Type II error*. Consequently, a false alarm may lead to a potentially poor data recovery, while a missed detection ignores an entire frame of data requiring retransmission [6].

Given these two types of errors, the performance of a detector can be characterized by two parameters; namely, the *probability of false alarm* (P_F), and the *probability of missed detection* (P_M) [7], which correspond to Type I and Type II errors, respectively, and thus can be defined as

$$P_F = P\{\text{Decide } \mathcal{H}_1 | \mathcal{H}_0\}, \tag{8.10}$$

and

$$P_M = P\{\text{Decide } \mathcal{H}_0 | \mathcal{H}_1\}. \tag{8.11}$$

Note that based on P_M, another frequently used parameter is the *probability of detection*, which can be derived as follows:

$$P_D = 1 - P_M = P\{\text{Decide } \mathcal{H}_1 | \mathcal{H}_1\}, \tag{8.12}$$

which characterizes the detector's ability to identify the primary signals in the channel, so P_D is usually referred to as the power of the detector.

As for detectors, we would like their probability of false alarm to be as low as possible, and at the same time, their probability of detection as high as possible. However, in a real-world situation, this is not achievable, because these two parameters are constraining each other. To show their relationship, a plot called *receiver operating characteristic* is usually employed [8], as shown in Figure 8.6, where its x-axis is the probability of false alarm and its y-axis is the probability of detection. From this plot, we observe that as P_D increases, the P_F is also increasing. Such an optimal point that reaches the highest P_D and the lowest P_F does not exist. Therefore, the detection problem is also a trade-off, which depends on how the Type I and Type II errors should be balanced.

When we consider the implementation consequences of detecting a signal, our design become more complicated than for example from Code 8.1. In the most basic sense detection becomes a thresholding problem for our correlator. Therefore,

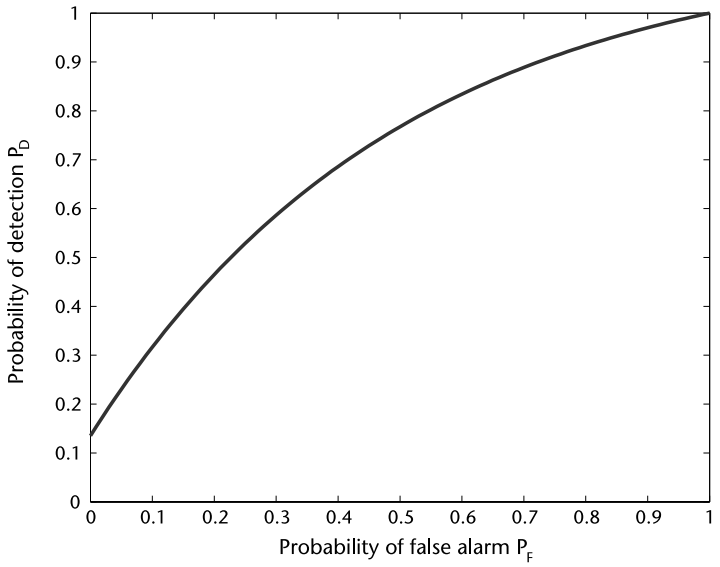


Figure 8.6 A typical receiver operating characteristic, where the x-axis is the probability of false alarm (P_F), and the y-axis is the probability of detection (P_D).

the objective becomes determining a reference or criteria for validating a peak, which can be radically different over time depending on channel noise and the automatic gain control of the Pluto SDR. However, even in simulations appropriate thresholding becomes nontrivial, which we can demonstrate with Figure 8.7(a) and 8.7(b). In these figures the peak appears larger relative to the rest of the correlation in the case where no frame exists in the receive signal compared to the condition when a frame exists. Therefore, for an implementation that performs well it should handle such conditions and operate regardless of the input scaling.

A common technique to aid with this thresholding process is to self-normalize the received signal. If we look back at Figure 8.7, we will notice that the magnitude can be quite different, which makes thresholding even more difficult. If we self-normalize the signal we can force it into a range closely between $\in [0, 1]$. A simple way to accomplish this operation is to scale our cross-correlation metric $C_{y,x}$ by the mean energy of the input signal x. To do this in an efficient way we can again utilize filtering to accomplish this task by implementing a moving average filter. Mathematically, this moving averaging would be modeled as another sum:

$$u_{ma}[n] = \sum_{i=0}^{N} u[n-i], \tag{8.13}$$

where N is the length of the preamble or sequence of interest. A useful aspect of (8.6) and (8.13) is that these are simple FIR filter implementations, making them simple to implement in hardware. In fact (8.13) requires no multiplication like the CIC filter discussed in Section 2.6.4. This aspect is import since in many systems this frame synchronize may also be used as packet detection mechanism at the front of the receiver, requiring it to run at the fastest rate of the input data without decimation. Combining our correlator from (8.6) and scaler from (8.13) we can

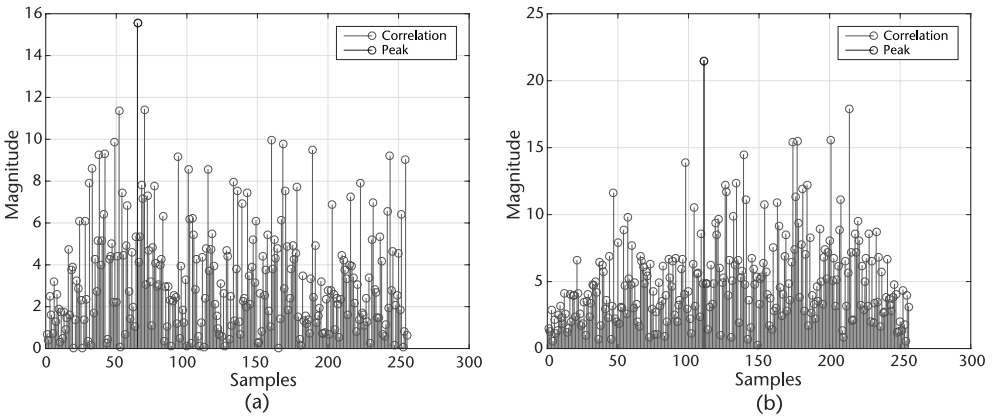


Figure 8.7 Example of false peaks in a cross-correlation sequence search. (a) Correlation without signal present, and (b) Correlation with signal present.

write our detector as

$$\mathcal{H}_0 : \frac{y[n]}{u_{ma}[n]} < T \text{ no signals},$$
$$\mathcal{H}_1 : \frac{y[n]}{u_{ma}[n]} \geq T \text{ signals exist}, \tag{8.14}$$

where T is our threshold value.

In MATLAB code `lst:findSignalStartTemplate` we have provided a template that nicely compensates for the transmit filter delay in the system, providing the true delay of a given packet at the receiver.

From the code provided in 8.2, implement a preamble start estimator using `xcorr` and the `filter` function. Evaluation the estimation accuracy over SNRs $\in [0, 12]$ dB in single dB steps.

8.2.2 Alternative Sequences

Besides Barker sequences, there are other sequences that have similar properties of minimal cross correlation except at specific instances. Two popular options are Zadoff-Chu sequences and Golay complementary sequences, which are currently both part of existing wireless standards.

Zadoff-Chu sequences, named after authors Solomon Zadoff and David Chu [9], are used for LTE synchronization and channel sounding operations. They are useful since they have a constant amplitude, zero circular autocorrelation, and very low correlation between different sequences. This properly of limited correlation between themselves is useful in a multiaccess environment where many users can transmit signals. Mathematically, the sequence numbers are generated as

$$s_n = exp\left(-j\frac{\pi\, k\, n\, (n + 1 + 2q)}{L} \right), \tag{8.15}$$

where L is the sequence length, n the sequence index, q and integer, and k, which is coprime with L. Unlike Barker sequences, which are purely integers 1 or -1, Zadoff-Chu sequences are complex valued.

Code 8.2 Loopback Pluto Example: **`findSignalStartTemplate.m`**

```
%% General system details
sampleRateHz = 1e6; samplesPerSymbol = 8; numFrames = 1e2;
modulationOrder = 2;  filterSymbolSpan = 4;
barkerLength = 26; % Must be even
%% Impairments
snr = 15;
%% Generate symbols and Preamble
bits = randi([0 3], modulationOrder*1e3,1);
hBCode = comm.BarkerCode('Length',7,'SamplesPerFrame', barkerLength/2);
barker = hBCode()>0; frame=[barker;barker;bits];frameSize = length(frame);
% Modulate
modD = comm.DBPSKModulator(); bMod = clone(modD);
modulatedData = modD(frame);
%% Add TX/RX Filters
TxFlt = comm.RaisedCosineTransmitFilter(...
    'OutputSamplesPerSymbol', samplesPerSymbol,...
    'FilterSpanInSymbols', filterSymbolSpan);
RxFlt = comm.RaisedCosineReceiveFilter(...
    'InputSamplesPerSymbol', samplesPerSymbol,...
    'FilterSpanInSymbols', filterSymbolSpan,...
    'DecimationFactor', samplesPerSymbol);
RxFltRef = clone(RxFlt);
%% Setup visualization object(s)
hts1 = dsp.TimeScope('SampleRate', sampleRateHz,'TimeSpan', ...
    frameSize*2/sampleRateHz);
hAP = dsp.ArrayPlot;hAP.YLimits = [-3 35];
%% Demodulator
demod = comm.DBPSKDemodulator;
%% Model of error
BER = zeros(numFrames,1);PER = zeros(numFrames,1);
for k=1:numFrames
    % Insert random delay and append zeros
    delay = randi([0 frameSize-1-TxFlt.FilterSpanInSymbols]);
    delayedSignal = [zeros(delay,1); modulatedData;...
        zeros(frameSize-delay,1)];
    % Filter signal
    filteredTXDataDelayed = TxFlt(delayedSignal);
    % Pass through channel
    noisyData = awgn(filteredTXDataDelayed,snr,'measured')
    % Filter signal
    filteredData = RxFlt(noisyData);
    % Visualize Correlation
    hts1(filteredData);pause(0.1);
    % Remove offset and filter delay
    frameStart = delay + RxFlt.FilterSpanInSymbols + 1;
    frameHatNoPreamble = filteredData(frameStart:frameStart+frameSize-1);
    % Demodulate and check
    dataHat = demod(frameHatNoPreamble);
    demod.release(); % Reset reference
    BER(k) = mean(dataHat-frame);PER(k) = BER(k)>0;
end
% Result
fprintf('PER %2.2fn',mean(PER));
```

The second sequence of interest are Golay complementary sequences, which are currently used in IEEE 802.11ad. Again they are used for channel estimation and synchronization within the preamble of IEEE 802.11ad packets. Golay complementary sequences are sequences of bipolar symbols with minimal autocorrelation properties. Therefore, they have a very similar to concept to Barker codes. However, as the name suggests these sequences come in complementary pairs that are typically denoted as Ga_n and Gb_n, where n is the sequence length. IEEE 802.11ad uses pairs Ga_32, Ga_64, and Gb_64. Using these sequences with BPSK is exceptional since performing the autocorrelations under even severe phase rotation is high. Another important aspect with Golay or specifically Ga and Gb sequence pairs is that their autocorrelation can be performed in parallel in hardware. This is very useful for a standard like 802.11ad, which is targeting transfer rates of 7 Gbits/s [10]. Building on this concept of minimal autocorrelation pairs and parallel processing of sequences, the preamble in IEEE 802.11ad can be used to provide signaling information to the receiver just based on its autocorrelation properties. This means that depending on the packet type a correlator bank can be used to identify that specific structure, conditioning the processing receiver to a specific decoder path for that type of packet.

8.3 Putting the Pieces Together

At this point we have all the necessary pieces to build a wireless receiver that can handle carrier offsets, timing mismatches, and random packet delay. With all these components in our tool belt, now it is a good time to talk about the system as a whole and determine the arrangement of components based on system requirements. This discussion on algorithm arrangements will be based on what we have learned from the previous chapters.

Starting at the front of the receiver we need to first accomplish two goals: carrier offset removal and timing synchronization with the received signal. In the system proposed so far we have first implemented timing recovery in the receive chain, but this requires usage of a TED, which is insensitive to phase rotation. Based on the options provided in Chapter 4, this would require a technique such as Gardner or a polyphase type implementation from Harris [11]. It is possible to utilize the FFC implementation described in Chapter 7 before timing recovery, but there can be residual phase noise left in the system. The receiver would be arranged similar to Figure 8.8. However, it is definitely useful to place a CFO before all processing, even before matched filtering, to reduce the work of other recovery algorithm in the receive chain. With that said, inserting CFO after a matched filter can make the estimates more accurate from CFO since the received signal will be SNR maximized. You must consider the trade-off in all these situations.

In Figure 8.9 we have outlined a possible receiver flow that contains the relative sample rates R_n between the recovery stages. The blocks with dashed outlines, the

Figure 8.8 Example receiver processing flow to recover transmitted frames where frequency recovery is considered first.

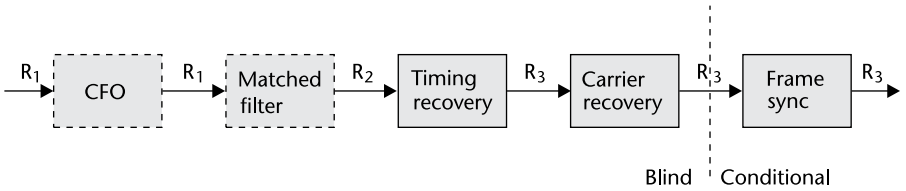


Figure 8.9 Complete receiver processing flow to recover transmitted frames. The relative sample rates are defined by R_n.

matched filter and CFO, can be optional if a RRC filter is used at the transmitter or the carrier offset is not severe. We specifically place the CFO before matched filtering since the matched filter can reduce the bandwidth CFO can utilize. With regard to the rates $R_m \geq R_k$ where $m < k$, meaning that the system will never upsample downstream or run at faster rates. Overall, downsampling will only occur in two possible stages: matched filtering and timing recovery. It is not required to do so in either stage but they can have specific benefits. For example, decimating at the matched filter stage will provide useful averaging, making symbols easier to distinguish. From a hardware perspective decimating reduces the sample rate and the constrains on downstream processing, lowering the bounds on clock rates for intensive operations in our downstream loops. When considering our timing recovery algorithms we already know from Chapter 6 that we can have specific requirements on R_n for the TED utilized. Providing the timing recovery loop with more samples per symbol can provide better performance as well.

In Figure 8.9 between the carrier recovery and frame sync block we have partitioned the system into two domains, blind and conditional. This is to define which processing blocks are dependent on the actual data transmitted and those blocks which are essentially blind to this fact. We can utilize this aspect of the design to introduce training data into our system that can be used to help the system converge to lock states in the recovery loops before actual data needs to be recovered. In this type of configuration we could prepend random bits that would be modulated and filtered as actual data of the transmitter to be sent. This extra or training data could just be prepended to the start of each frame or continuously transmitted in between data frames to keep the receiver locked. This would remove convergence delays in our system. In hardware we could simply connect a linear-feedback shift register (LFSR) to our modulator, which is a convenient mechanism of generating random bits. In MATLAB this is represented by the `comm.PNSequence` System object. Once converged it may not be necessary to continuously transmit training data, which would increase the throughput of the system.

When implementing a system with Pluto SDR or even in simulation it can be helpful to introduce training information into your system. In this configuration our frame sync block from Figure 8.9 would act as a gateway to downstream processing, only allowing data to pass through once a preamble or marker was detected.

8.3.1 Full Recovery with Pluto SDR

Throughout the last three chapters we have introduced templates of code and provided guidance on how to implement scenarios with Pluto SDR. However, when considering full frame recovery and actually demodulation of data we need

to reenforce some implementation details. Since the receiver is a complex system requiring many processing components, real-time operation should not be an initial goal. Once your receiver algorithms are working you can extend them to work in real-time if desired. Therefore, for your implementations you should focus on coding templates Code 5.3, 5.4, and 5.5. Performing processing in between calls to Pluto SDR similar to Code 5.6 will likely result in overflows and missed data, making it difficult to recover full frames of data.

A second important tool to utilize is the `transmitRepeat` method of Pluto SDR as in code example 5.7. This will deterministically transmit data continuously. With regard to this data at the receiver, since the delay will be random from the transmitter, you should always collect at least $2L$ samples where L is the length of the desired frame. Setting Pluto SDR's `SamplesPerFrame` property to $2L$ will guarantee at least one full packet received when the transmitter is in `transmitRepeat` mode. This was unnecessary in Chapters 4 and 7 since we could lose data to an overflow and this would have little impact on our tests. However, this is paramount when checking for full frames. Here is a simple example to follow in Code 8.3. In this example we actually step the receive several times to remove possible stale data in its IIO buffers.

Code 8.3 Capture Repeated Frame: **`captureExample.m`**

```
% Transmit frame repeatedly
tx = sdrtx('Pluto');
tx = sdrtx('Pluto','SamplesPerFrame',length(frame)*2);
tx.transmitRepeat(frame);
for k=1:4,rx();end; % Remove stale data from buffers
rxBuffer = rx();
```

Using the template from Code 8.3 and the synchronization blocks developed in Chapters 4, 7, and in this chapter, begin to estimate the start of packets. First, utilizing `transmitRepeat` collect $L \times N$ samples of data where $N = 1, 2, 3, 4, 5$. Your packet detector should be able to locate at least $L \times (N - 1)$ of data for each iteration of N. Evaluate the performance of your system for this growing number of packets. Collect 1,000 packets and determine your probability of detection (try to collect 10 packets at a time and repeat this process).

A useful tool when evaluating complete recovered frames is a cyclic redundancy check (CRC) that provides a binary value if a packet contains errors or not. Conveniently, MATLAB's Communication Systems Toolbox contains a system object with this functionality called `comm.CRCGenerator` and its mirrored detector `comm.CRCDetector`. They can be called in the following way in Code 8.4 where we utilize the polynomial $z^3 + 1$ in both objects.

A CRC works by appending a sequence of length L_c, called a checksum, to the end of the our data. L_c will be equal to the order of the polynomial, which is

Code 8.4 Loopback Pluto Example: **`crcExample.m`**

```
x = logical([1 0 1 1 0 1 0 1 1 1 0 1]');
crcGen = comm.CRCGenerator('z^3 + 1');
crcDet = comm.CRCDetector('z^3 + 1');
codeword = crcGen(x);
codewordWithError = codeword; codewordWithError(1) = ~codewordWithError(1);
[tx, err]   = crcDet(codeword);
[tx1, err1] = crcDet(codewordWithError);
```

three in the case of Code 8.4. This polynomial determines how bits are combined (XORed) together to produce the checksum. The larger the value L_c the lower the probability of Type I or Type II errors, as outlined in Section 8.2.1. However, this is also dependent on the length of the data related to the checksum. In practice, the data related to a checksum size will be orders of magnitude greater than L_c. With regard to our frame synchronization testing we can utilize CRCs in transmitted frames to easily check if we have recovered all our transmitted bits.

Again, using the template from Code 8.3, and the synchronization blocks developed in Chapters 4, 7, and in this chapter, begin to estimate the start of packets. This appends CRC to each frame before transmission. At the receiver demodulate the recovered symbols, check the CRC values for 1,000 packets. Repeat this process but calculate the bit error rate for each frame recovered. Skip lost frames.

8.4 Channel Coding

Now that we can successfully recover data across the wireless link, we can discuss techniques of making this process more robust. Channel coding is an obvious option and is ubiquitous in any digital communications standard.

8.4.1 Repetition Coding

One of key building blocks of any communication system is the forward error correction (FEC), where redundant data is added to the transmitted stream to make it more robust to channel errors. There are many types of FEC techniques, such as the *repetition coding* approach, where each transmitted bit is repeated multiple times. In this section, we will explore together one technique for combating the introduction of errors to data transmissions by implementing a simple repetition coder (repetition factor $R = 4$). So what does it mean by a repetition coder with repetition factor $R = 4$? A simple definition is that if a "0" symbol is to be transmitted, this "0" symbol will be repeated four times by the repetition coder, such that the output would be "0000."

Let us first start by double-clicking on the repetition coder block, which will result in a MATLAB function block editor opening, in which we can write

customized MATLAB code. As mentioned previously, setting break points is a great way for understanding and debugging M-files. For more information about break points and how they can be used to debug and evaluate the code, please refer to Appendix B.

The *repmat* function in MATLAB can be used to realize a simplistic repetition coding scheme. For example, to repeat a vector u for 4 times, the following expression can obtain this result:

```
y=repmat(u,4,1);
```

What are the trade-offs to consider when choosing between a high or a low repetition factor?

8.4.2 Interleaving

A repetition code is one of several useful tools for a communication systems engineer in order to enhance a robust data transmission. However, it is sometimes not enough, since it does not address the issue when a large quantity of data is corrupted in contiguous blocks. For instance, if a transmitter sends the data stream "101101," a repetition coder with a repetition factor of 4 will yield

$$111100001111111100001111,$$

where each input bit is repeated four times. While this encoding scheme may appear robust to error, it is still possible during a data transmission that a significant noise burst occurs over many consecutive bits, corrupting numerous binary digits in the transmission, and yields the following outcome:

$$111100 - - - - - - - - 1100001111,$$

where some of the original data is completely irretrievable.

Why is it that even with repetition coding, our data transmission can still be severely affected? What could be done to make it even more robust?

Interleaving is an approach where binary data is reordered such that the correlation existing between the individual bits within a specific sequence is significantly reduced. Since errors usually occur across a consecutive series of bits, interleaving a bit sequence prior to transmission and deinterleaving the intercepted sequence at the receiver allows for the dispersion of bit errors across the entire sequence, thus minimizing its impact on the transmitted message. A simple interleaver will mix up the repeated bits to make the redundancy in the data even more robust to error. It reorders the duplicated bits among each other to ensure that at least one redundant copy of each will arrive even if a series of bits are lost. For

example, if we use an interleaving step of 4, it means we reorder the vector by index [1, 5, 9, ..., 2, 6, 10, ...]. As a result, running "111100001111111100001111" through such an interleaver will yield the following output:

101101101101101101101101.

The interleaving step can be any of the factoring numbers of the data length. However, different mixing algorithms will change the effectiveness of the interleaver.

The *reshape* function in MATLAB can be used to realize the interleaving.

Once we have implemented the interleaver, let us combine the repetition coder and the interleaver into a single FEC subsystem. Although the simple interleaving technique introduced above is sufficient for our implementation, there are various other forms of interleaving, that we will investigate in the next two sections.

8.4.2.1 Block Interleaving

The first approach to interleaving is to employ a block interleaver, as shown in Figure 8.10. Block interleaving is one method for reordering a bit sequence, where $N \times M$ bits fill an N column by M row matrix on a column basis, and then each resulting row is concatenated with each other in serial and outputted from the interleave block. At the transmitter side, the block interleaver is loaded column by column with N codewords, each of length M bits. These N codewords are then transmitted row by row until the interleaver is emptied. Then the interleaver is loaded again and the cycle repeats. The main drawback of block interleavers is the delay introduced with each column-by-column fill of the interleaver [12].

8.4.2.2 Convolutional Interleaving

Another approach to interleaving is to employ a convolutional interleaver [13], as shown in Figure 8.11. At the transmitter, the bit sequence is shifted into a bank of N registers, each possessing an increasing amount of buffer memory. The bits in the bank of registers are then recombined via a commutator and transmitted across the channel. At the receiver, the reverse process is performed in order to recover the original sequence. Compared with block interleavers, convolutional interleavers reduce memory requirements by about one-half [14]. However, the delay problem associated with the initial fill still exists.

8.4.3 Encoding

Besides interleaving multiple copies of data, we can instead encode the data into alternative sequences that introduce redundancy. A unique property of many encoding schemes is the ability to introduce redundancy without increases in data size without integer order. For example, in the case of repetitive coding that duplicates every bit with $R = 2$, this number is usually inverted in FEC discussions as a rate of $\frac{1}{2}$, a convolutional encoding scheme can introduce rates closer to 1. This makes them more efficient and provides more effective throughput. In this

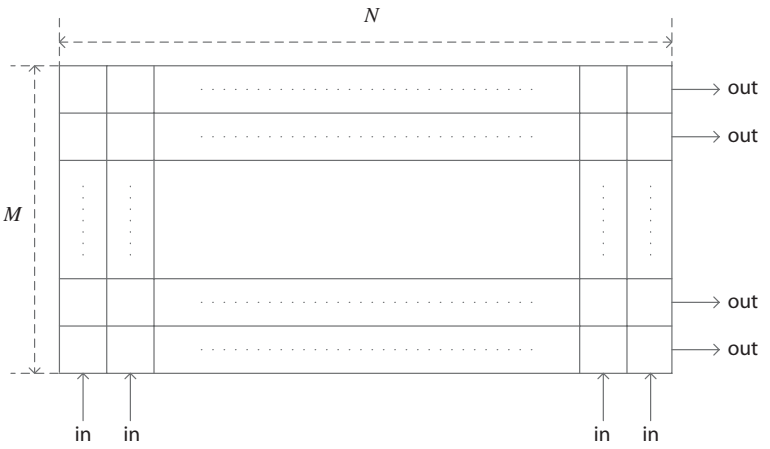


Figure 8.10 Schematic of a block interleaver.

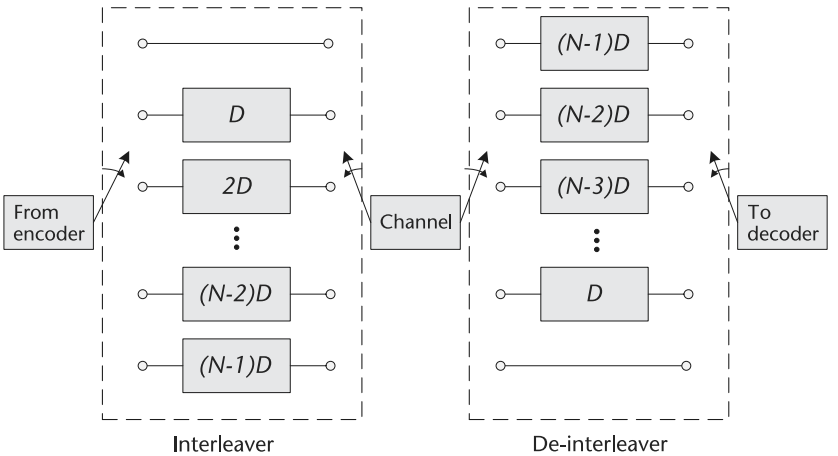


Figure 8.11 Schematic of a convolutional interleaver. (From [13].)

section we will discuss several common channel encoding schemes, with some basic information on how they function. In general, channel encoding is a mathematically complex area in information theory. Instead of diving into the theoretical designs of these scheme, we will compare their relative performance as well as some implementation details and drawbacks.

Similar to interleavers, encoders can typically be categorized into two basic types: block encoders and convolutional type encoders. Block encoders work on specific predefined groups or blocks of bits. Alternatively, convolutional encoders work on streams of data of indeterminate size but can be made to work on blocks of data if necessary.

The first coding scheme we will discuss is Reed-Solomon (RS) codes, which are linear-block-code developed in the 1960s. RS codes work by inserting symbols into a given frame or block of data, which are then used to correct symbol errors that occur. If we define M as the length of a given frame, sometimes called the message length, and define E as the encoded frame then we can correct up to $\lfloor \frac{E-M}{2} \rfloor$ symbols. RS are interesting since they can even provide information on how many errors were found and corrected as part of their implementation. However, RS encoders can be specific on what inputs they can process, which is why we have so far only

considered symbols not bits. The symbols you can encode with a RS can be integers between $[0, 2^N - 1]$, where N is the exponent of our finite Galois field $GF(2^N)$. A Galois field is a field, a set which certain mathematical operations are defined, which has a finite number of objects. Due to this definition there will be some restrictions on E and N, but they are beyond the scope of this book. This set is how RS takes advantage of during decoding, which will reduce solutions spaces based on received data and selection of M, E and N.

With regard to implementation, it can be argued that RS codes are useful in bursty error situations where a swath of symbols close to each other are corrupted. When considering transmitted bits, each symbol will represent B bits, and since RS operate on symbols it can correct B bits of data in a group. Therefore, a bursty error corrupting $B + 1$ bits in a group can corrupt at most 2 symbols.

A similar code to RS is Bose Chaudhuri Hocquenghem (BCH) codes, which also relies on the concept of Galois fields. BCH codes are better at correcting errors that do not occur in groups, unlike RS. To reduce this probability of grouped data it can be useful to shuffle or scramble bits before and after transmission to reduce error locality, which is better for BCH when errors are sparse. However, this is a poor thing to do with RS to some extent. BCH codes can also correct more errors for the same amount of parity bits, but in general BCH codes require more computational power to decode than RS.

The final popular block code to consider are low-density parity check (LDPC) codes, which have even begun to replace the dominant Turbo codes, which we will consider next. LDPC codes have been around since the 1960s, but due to their complexity have only been considered for hardware implementation in the last decade. Developed by Robert Gallager, LDPC codes can approach the theoretical Shannon limit [15] for certain redundancy rates unlike RS and BCH. However, the computation required to use LDPC is considerable higher. Nonetheless, they exist in some modes of 802.11n and DVB-S2 standards.

When utilizing LDPC the implementor must select a parity matrix which the encoder and decoder will utilize, and the characteristics of this matrix will determine performance of the code. Strong performing codes will typically come in large block lengths like 648, 1296, and 1944 IEEE 802.11n/ac. This means that, you need to encode a significant amount of bits compared to the other codes to utilize LPDC efficiently in many cases.

Besides block codes, an alternative or stream-based coding implementation is convolutional codes. These codes convolutionally encode data, meaning redundancy is introduced by the succession of information passed through the encoder/decoder, essentially creating dependency on consecutive symbols or bits. A convolutional encoder is best understood by an example. Let us consider an encoding scheme with $R = 2$ with a recursive encoder, with three registers. Figure 8.12 provides a possible structure for such an encoder, which outputs two bits for every bit pushed into the system. You will notice that the current output is at least dependent on the last three inputs, similar to the memory of an FIR or IIR filter.

Figure 8.12 can be interpreted as two equations, provided in (8.16).

$$\begin{aligned} y_{n,1} &= (x_n + x_{n-2} + x_{n-3}) + x_{n-1} + x_{n-3} \\ y_{n,2} &= x_{n-2} + x_{n-3} \end{aligned} \tag{8.16}$$

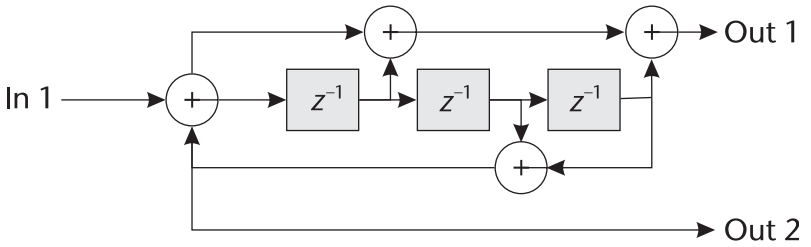


Figure 8.12 Example $R = 2$ convolutional encoder utilized in 3GPP LTE.

The decoder itself will utilize this dependency in the data to help remove errors that can occur. The most popular algorithm to accomplish this task is called the Viterbi algorithm [15], sometimes called a trellis algorithm or decoder. The concept of the Viterbi/trellis decoder is to trace back through previous decisions made and utilize them to best determine the most likely current bit or sample. This is what naturally leads to Figure 8.13 and hence the name trellis. In Figure 8.13, the left-most position represents the most recently receiver symbols or bits. The lines connecting the dots represent possible previous symbols, where the thick gray line represents the more probable symbols based on previous decisions made. The depth of this trellis is called the traceback length, and the deeper this trace becomes the better the recovery of bits will be. However, this process tends to plateau at a traceback around 34 symbols, and the deeper the traceback the increased time required to decode a specific symbol. This traceback limitation can be seen in Figure 8.14, where we examine a variety of traceback lengths across EbN0 for a 16-QAM signal.

In the early 1990s turbo codes were introduced, which are part of the convolutional code family [16]. Turbo codes have been heavily utilized by both third and fourth generation cellular standards as their primary FEC scheme. Like LDPC, turbo code can operate near the Shannon limit for performance but are less computationally intensive than LDPC with less correction performance. Turbo inherently utilizes the Viterbi algorithm internally for decoding with some additional interleaving, as well as using a set of decoders, and performs likelihood estimation between them. This is an extreme simplification of how turbo decoders actually work, and analysis of their operation is a very difficult topic area. However, they are a very powerful coding technique as long as you have the resources on your hardware to implement the decoder at the necessary speeds.

When utilizing FEC one should always consider the trade-offs with regard to computational complexity, performance required, and coding overhead allowed for the link. This is important since heavy coding may not be required in a clear transmission channel where additional throughput could be gained at a better coding rate closer to one. Therefore, modern standards like LTE and IEEE 802.11, will utilize adaptive modulation and coding schemes (MCSs), which reduce coding redundancy and increase modulation order, providing much higher throughput across a link. IEEE 802.11 itself has 32 MCS states or indexes, for which we have provided the first four entries in Table 8.2, for perspective on how code rates and modulation are used to trade off redundancy and data rate.

MATLAB itself provides all of the coding techniques we have described so far. However, some of the advanced codes are more complex to utilize, especially in

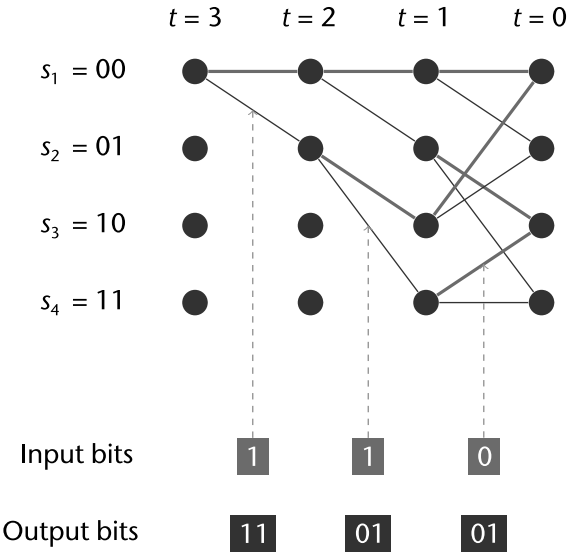


Figure 8.13 Viterbi/trellis decoder lattice diagram.

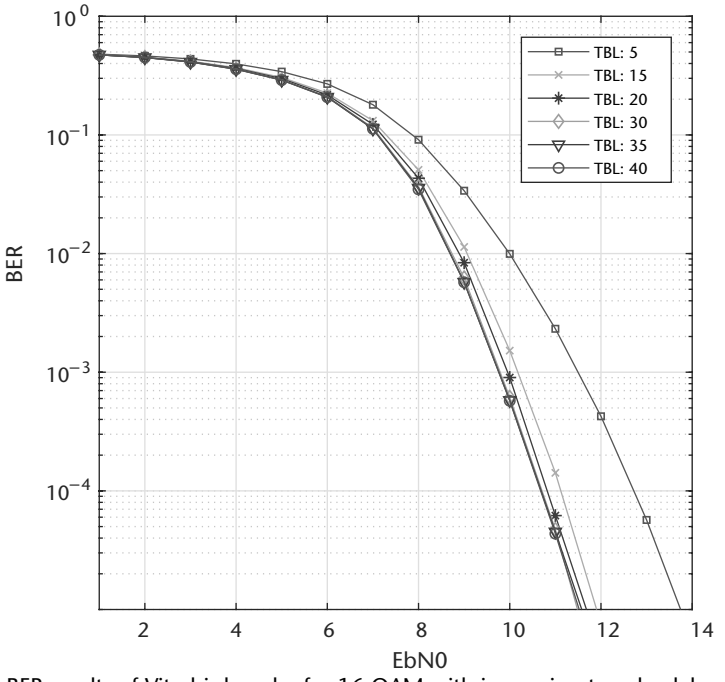


Figure 8.14 BER results of Viterbi decoder for 16-QAM with increasing traceback length.

different modes. For example, due to how turbo decoders work they require channel estimates to correctly determine the noise variance of the channel. Therefore, in a give receiver design this information must be provided for effective decoding. LDPC, as we have discussed before, requires parity matrices in their design. By default MATLAB provides the parity matrix for DVB, which requires 32,000 bits per block, which is rather unreasonable for many applications. Designing a smaller matrix can be complex, and this is rather outside the scope of MATLAB's provided

Table 8.2 Shortended Modulation and Coding Schemes List for IEEE 802.11*

MCS Index	*Streams*	*Modulation*	*R*	*Data Rate (Mbits/s)*
0	1	BPSK	2	6.5
1	1	QPSK	2	13
2	1	16-QAM	4/3	19.5
3	1	16-QAM	2	26

* From [1]

tools. Nonetheless, RS, BCH, and general Viterbi decoding are extremely easy to utilize out of the box and are simple to parameterize.

8.4.4 BER Calculator

After examing several techniques for protecting the reliability of a data transmission subject to interference and noise, we now need an approach to measure how well these techniques perform quantitatively. Referring back to Chapter 4, we saw that BER is a commonly used metric for the evaluation and comparison of digital communication systems. One straightforward way of calculating the BER is to count the number of received bits of information and then determine which ones are received in error. In other words, the ratio of bit errors to the total number of bits received can provide us with an approximate BER calculation. Note that the more bits that are received, the more accurate this link level metric becomes.

8.5 Chapter Summary

This chapter examined the concept of frame synchronization through correlation techniques and covered some common channel coding techniques through redundancy insertion. Given the last piece of the receiver with regard to synchronization provided here, a full receiver can be implemented for frame recovery. Possible arrangement for the receiver algorithms as discussed throughout the book so far have been examined, focusing on requirements from the design. Once full synchronization was covered, we moved on to techniques for making our links more robust through encoding implementations, including a discussion on their drawbacks and advantages with regard to implementation and use.

References

[1] IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications–Amendment 4: Enhancementsfor Very High Throughput for Operation in Bands below 6 GHz, IEEE Std 802.11ac-2013, (Amendment to IEEE Std 802.11-2012, as amended by IEEE Std 802.11ae-2012, IEEE Std 802.11aa-2012, and IEEE Std 802.11ad-2012), December 2013, pp. 1–425.

[2] Barke, R. H., "Group Synchronizing of Binary Digital Sequences," in *Communication Theory*, London: Butterworth, 1953, pp. 273–287.

[3] The Math Works Inc., xcorr [online], 2017 https://www.mathworks.com/help/signal/ref/xcorr.html.

[4] Kay, S. M., *Fundamentals of Statistical Signal Processing*, Volume II: Detection Theory. Upper Saddle River, NJ: Prentice Hall, 1998.

[5] Poor, H. V., *An Introduction to Signal Detection and Estimation*, New York: Springer, 2010.

[6] Zhao, Q., and A. Swami, "Spectrum Sensing and Identification" in *Cognitive Radio Communications and Networks: Principles and Practice*, Burlington, MA: Academic Press, 2009.

[7] Kay, S. M., "Statistical Decision Theory I," in *Fundamentals of Statistical Signal Processing*, Volume II: Detection Theory, Upper Saddle River, NJ: Prentice Hall, 1998.

[8] Shanmugan, K. S., and A. M. Breipohl, "Signal Detection," in *Random Signals: Detection, Estimation and Data Analysis*, Wiley, 1988.

[9] Finger, A., *Pseudo Random Signal Processing: Theory and Application*, Hoboken, NJ: Wiley, 2013.

[10] IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems–Local and Metropolitan Area Networks–Specific Requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 3: Enhancements for Very High Throughput in the 60 GHz Band, IEEE Std 802.11ad-2012 (amendment to IEEEStd 802.11-2012, as amended by IEEE Std 802.11ae-2012 and IEEE Std 802.11aa-2012), December 2012,pp. 1-628.

[11] Harris, F. J., and M. Rice, "Multirate Digital Filters for Symbol Timing Synchronization in Software Defined Radios," *IEEE Journal on Select Areas in Communications*, Vol. 19, October 2001, pp. 2346–2357.

[12] Jacobsmeyer, J. M., Introduction to Error-Control Coding, www.pericle.com/papers/Error_Control_Tutorial.pdf.

[13] Forney, G. D., "Burst-Correcting Codes for the Classic Bursty Channel, in *IEEE Transactions on Communications, COM-19*, 1971, pp. 772–781, 1971.

[14] Sklar, B., *Digital Communications Fundamentals and Applications*, Upper Saddle River, NJ: Prentice Hall, 1988.

[15] Anderson, J., and S. Mohan, *Source and Channel Coding: An Algorithmic Approach*, New York: Springer, 1991.

[16] Berrou, C., Error-Correction Coding Method with at Least Two Systematic Convolutional Codingsin Parallel, Corresponding Iterative Decoding Method, Decoding Module and Decoder, US Patent No. 5,446,747, 1995.