Fully Configurable OFDM SDR Transceiver in LabVIEW

Collection Editors:

Bryan Paul

Christopher Schmitz

Fully Configurable OFDM SDR Transceiver in LabVIEW

Collection Editors:
Bryan Paul
Christopher Schmitz

Authors:
Aditya Jain
Bryan Paul

Online:
< http://cnx.org/content/col11182/1.6/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Chapter 1

Overview

1.1 Introduction to OFDM and SDRs[1]

OFDM (Orthogonal Frequency Division Multiplexing) is the technique for transmitting data in parallel using large number of modulated carriers with harmonic frequency spacing so that the carriers are orthogonal to each other. The orthogonality allows spectral overlapping of channels that can be separated later, much like quadrature modulation.

SDR (Software Defined Radio) refers to a radio communication system that can be configured to send and receive a wide range of modulated digital signals across a large frequency spectrum by means of a programmable hardware platform. With the advancements made in low noise amplifiers (LNAs), analog-to-digital converters (ADCs) or samplers, and antenna technology, SDRs have become an emergent technology in communications. This will allow the same antenna or antenna array to be used for different frequency ranges, and to move the ADC as close as possible to the antenna with little or no pre-filtering, and performing the entire signal processing digitally.

In a nut shell, OFDM is a modulation scheme that rides on top of another basic type of modulation such as BPSK and QAM to allow simultaneous transmission of independent signal carriers. It is highly scalable, allowing expansion or reduction of the signal bandwidth to accommodate the dynamic creation or removal of signal carriers. As a result these unique properties, it is widely used in a variety of important applications such as mobile radio and digital broadcasting. It has also been touted as the possible scheme of choice for the predicted paradigm shift known as the cognitive radio. In our Digital Signal Processing Laboratory project, we gained a firm grasp of this exciting technology by creating an OFDM transmitter and implementing a corresponding OFDM software receiver. The transmitter modulates BPSK signals and outputs them through a communications FPGA, while the receiver uses Matlab libraries to process the captured signals to retrieve the original data. At the end of the project, the final system was shown to be able to correctly transmit and receive 64 independent signal carriers simultaneously.

The new project, an independent study carried on by one of the three original students with Dr. Christopher Schmitz of the University of Illinois at Urbana/Champaign, will be an all LabVIEW transceiver implemented in actual hardware. It will run on a front-end receiver utilizing National Instruments' PXI chassis populated with the 5660 digital downconverter/high-speed digitizer in conjunction with the 5671 AWG/upconverter. The goal is a real-time system that transmits and receivers over the air, so that parameters can actively be tweaked, and the resulting changes in performance can be observed.

[1]This content is available online at <http://cnx.org/content/m33640/1.3/>.

Chapter 2

Transmitter

2.1 MATLAB

2.1.1 Bits-to-Words (Transmitter) (MATLAB)[1]

```
%% Bits to Words
function words = b2w(bits, bits_per_symbol)
%%
    num_subcarriers = size(bits,2)/bits_per_symbol;
    words = zeros(num_subcarriers, bits_per_symbol);

    for n=1:num_subcarriers
        words(n,1:bits_per_symbol) = bits((n-1)*bits_per_symbol+1:(n-1)*(bits_per_symbol)+bits_per_symb
    end
end
```

2.1.2 Words-to-Symbols (Transmitter) (MATLAB)[2]

```
%% Words to Symbols
function symbols = w2sym(words, word_to_symbol_map, bits_per_symbol)
%%
    num_subcarriers = size(words,1);
    symbols = zeros(1,num_subcarriers);

    for n=1:num_subcarriers
        symbols(1,n) = word_to_symbol_map(binvec2dec(fliplr(words(n, 1:bits_per_symbol)))+1);
    end
end
```

[1]This content is available online at <http://cnx.org/content/m34139/1.1/>.
[2]This content is available online at <http://cnx.org/content/m34142/1.1/>.

2.1.3 Symbols-to-FFT (Transmitter) (MATLAB)[3]

```
%% Symbols to FFT
function [re im] = sym2fft(symbols, size_of_fft)
%%
    num_subcarriers = size(symbols,2);
    input = [ 0 symbols(1:(num_subcarriers/2)) zeros(1,size_of_fft-num_subcarriers-1) symbols((num_subca
    output = ifft(input);
    re = real(output);
    im = imag(output);
end
```

2.1.4 Add Cyclic Prefix (Transmitter) (MATLAB)[4]

```
%% Symbols to FFT
function [re_cyc im_cyc] = cyc(re, im, perc_cyc)
%%
    size_of_fft = size(re,2);
    size_of_cyc = round(size_of_fft*(1-perc_cyc))+1;
    re_cyc = [re(size_of_cyc:size_of_fft) re];
    im_cyc = [im(size_of_cyc:size_of_fft) im];
end
```

2.1.5 OFDM Symbol Generator (Transmitter) (MATLAB)[5]

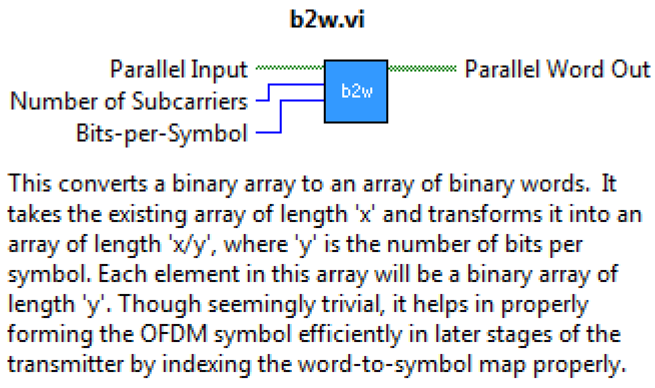
```
%% Generate Baseband OFDM Signal
function [re im] = ofdm_gen(bits, bits_per_symbol, word_to_symbol_map, size_of_fft, perc_cyc)
%%
words = b2w(bits,bits_per_symbol);
symbols = w2sym(words,word_to_symbol_map,bits_per_symbol);
[re_0 im_0] = sym2fft(symbols,size_of_fft);
[re im] = cyc(re_0,im_0,perc_cyc);
end
```

[3]This content is available online at <http://cnx.org/content/m34141/1.1/>.
[4]This content is available online at <http://cnx.org/content/m34138/1.1/>.
[5]This content is available online at <http://cnx.org/content/m34140/1.1/>.

2.2 LabVIEW

2.2.1 Bits-to-Words (Transmitter) (LabVIEW)[6]

Input/Outputs and Help

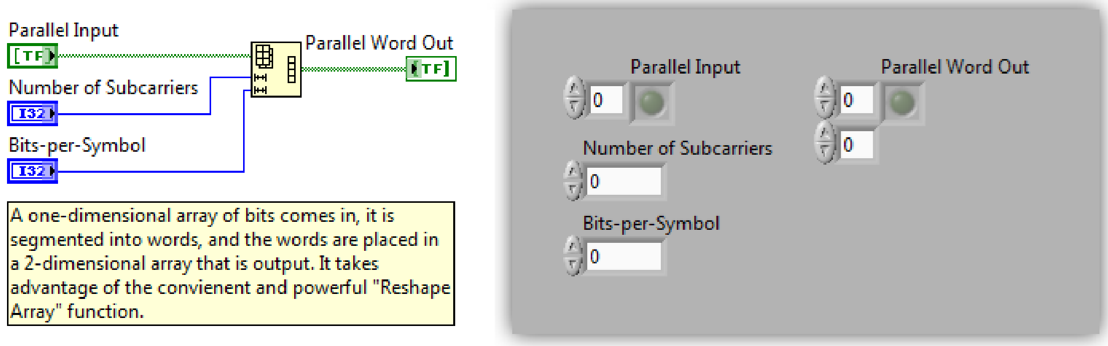


This converts a binary array to an array of binary words. It takes the existing array of length 'x' and transforms it into an array of length 'x/y', where 'y' is the number of bits per symbol. Each element in this array will be a binary array of length 'y'. Though seemingly trivial, it helps in properly forming the OFDM symbol efficiently in later stages of the transmitter by indexing the word-to-symbol map properly.

Figure 2.1: This is the LabVIEW help and block description for the Bits-to-Words sub- VI.

The sub-VI for the rather trivial operation of converting a string of bits to bit-words is shown above in Figure 1. Only basic knowledge of LabVIEW is required to implement this module. While some may argue it's trivial and non-deserving of its own sub-VI, modularity and readability still prevail in good design technique, and this transmitter is no exception.

Block Diagram Layout



A one-dimensional array of bits comes in, it is segmented into words, and the words are placed in a 2-dimensional array that is output. It takes advantage of the convienent and powerful "Reshape Array" function.

Figure 2.2: This is the LabVIEW block diagram for the Bits-to-Words sub-VI.

Figure 2 above shows the block diagram layout. The parallel bits are taken as an input and fed into the built-in LabVIEW function "Reshape Array," which groups the number of bits appropriately.

[6]This content is available online at <http://cnx.org/content/m34120/1.5/>.

For the logisitics in actually constructing this function, see the video below. For any additional questions, the example usage video in Figure 4, or email the author.

Instructional Video

This media object is a Flash object. Please view or download it at <http://www.youtube.com/v/JGGlo6h3SYA&hl=en&fs=1&rel=0>

Figure 2.3: This is the instructional video for constructing the Bits-to-Words sub-VI.

Example Video

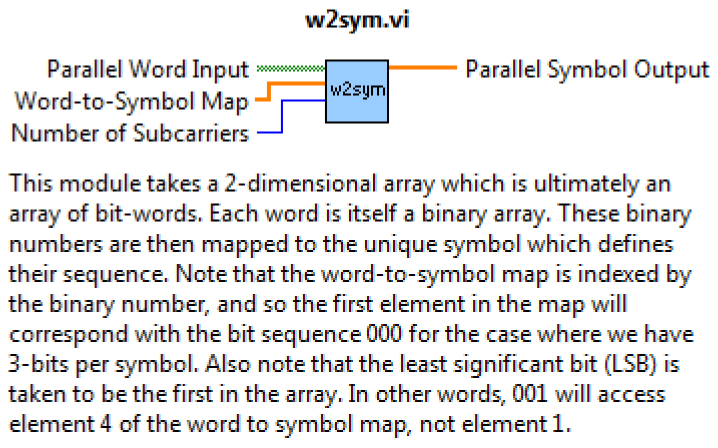
This media object is a Flash object. Please view or download it at <http://www.youtube.com/v/Bth8nkZ9C5A&hl=en&fs=1&rel=0>

Figure 2.4: This is the example video for using the Bits-to-Words sub-VI.

Download This LabVIEW sub-VI[7]

2.2.2 Words-to-Symbols (Transmitter) (LabVIEW)[8]

Input/Outputs and Help



This module takes a 2-dimensional array which is ultimately an array of bit-words. Each word is itself a binary array. These binary numbers are then mapped to the unique symbol which defines their sequence. Note that the word-to-symbol map is indexed by the binary number, and so the first element in the map will correspond with the bit sequence 000 for the case where we have 3-bits per symbol. Also note that the least significant bit (LSB) is taken to be the first in the array. In other words, 001 will access element 4 of the word to symbol map, not element 1.

Figure 2.5: This is the LabVIEW help and block description for the Words-to-Symbols Sub-VI.

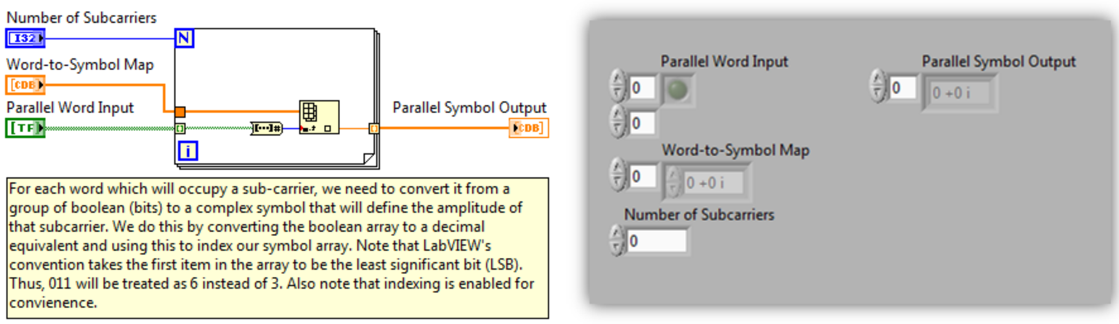
The above Figure (Figure 1) shows the Words-to-Symbols function. This is fairly straightforward: in band limited channels, we cannot send a square-wave stream of 1's and 0's. More importantly, we'd like to leverage

[7]http://cnx.org/content/m34120/latest/b2w.zip
[8]This content is available online at <http://cnx.org/content/m34127/1.5/>.

OFDM's attractive properties and transmit many data streams on multiple carriers.

Thus we must take each word (itself a group of bits), and assign it a unique complex signal. This is where any popular scheme is chosen such as BPSK, M-ary QAM, etc. By simply defining the word-to-symbol map, the numbers the words represent in the base 10 (decimal) system index elements in the map and the conversion is done.

Block Diagram Layout



For each word which will occupy a sub-carrier, we need to convert it from a group of boolean (bits) to a complex symbol that will define the amplitude of that subcarrier. We do this by converting the boolean array to a decimal equivalent and using this to index our symbol array. Note that LabVIEW's convention takes the first item in the array to be the least significant bit (LSB). Thus, 011 will be treated as 6 instead of 3. Also note that indexing is enabled for convienence.

Figure 2.6: This is the LabVIEW block diagram for the Words-to-Symbols sub-VI.

Figure 2 shows this simple process. As an example, let's say the bit-word is 110. LabVIEW's convention would read this as decimal 3, as the first item in the array is seen as the least significant bit (LSB). Next, we index our symbol array with decimal 3. Thus, entry 3 of our word-to-symbol map (which better have 8 unique entries in this example) will be the symbol representing the bit stream 110, say 4+4i. This complex amplitude is then stored in the output stream of parallel words, eventually to occupy an FFT bin.

The only potential pitfall here is the use of indexing. When an array is fed into a for loop, you can right click the entry point and enable indexing. This will present inside the loop the element corresponding to that iteration of the loop. In other words, the first time around, we're going to access the first bit-word (0th element), the second time around the second word etc. In that vein, we also build our output array in the same iterative fashion by indexing it as well.

If you are still confused on how this works, check out the tutorial video below, the example usage video in Figure 4 or email the author for more questions. The sub-VI is available below for download.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/ScB9H3OQpWA&hl=en&fs=1&rel=0>

Figure 2.7: This is the instructional video for constructing the Words-to-Symbols sub-VI.

Example Video

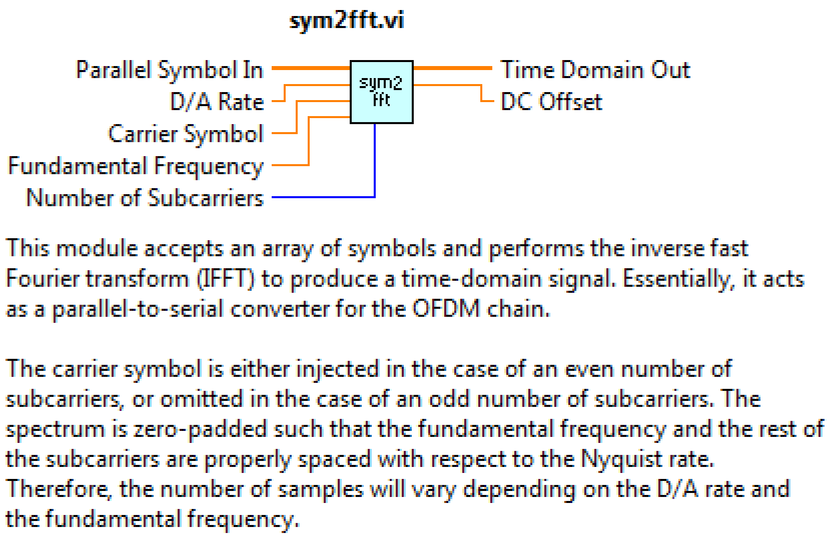
This media object is a Flash object. Please view or download it at <http://www.youtube.com/v/l_pRPWRVvy0&hl=en&fs=1&rel=0>

Figure 2.8: This is the example video for using the Words-to-Symbols sub-VI.

Download This LabVIEW sub-VI[9]

2.2.3 Symbols-to-FFT (Transmitter) (LabVIEW)[10]

Input/Outputs and Help



This module accepts an array of symbols and performs the inverse fast Fourier transform (IFFT) to produce a time-domain signal. Essentially, it acts as a parallel-to-serial converter for the OFDM chain.

The carrier symbol is either injected in the case of an even number of subcarriers, or omitted in the case of an odd number of subcarriers. The spectrum is zero-padded such that the fundamental frequency and the rest of the subcarriers are properly spaced with respect to the Nyquist rate. Therefore, the number of samples will vary depending on the D/A rate and the fundamental frequency.

Figure 2.9: This is the LabVIEW help and block description for the Symbols-to-FFT Sub-VI.

This function (shown above in Figure 1) implements the most important and often the most confusing part of the OFDM transmitter chain: converting the parallel group of symbols to a continuous time-domain signal. Note that this module requires the Digital-to-Analog coverter Rate (D/A Rate) and the fundamental frequency of the subcarriers. Both of these parameters were not needed until now, because we've been discretely preparing our OFDM data.

The most frequently asked question about OFDM is "Why IFFT?" To those unfamiliar, IFFT stands for "Inverse Fast Fourier Transform." This is simply an efficient algorithm for performing the Inverse Discrete Fourier Transform, the sampled version of the IDTFT (Inverse Discrete Time Fourier Transform) which is the Fourier Transform for discretized signals. Confused yet? Don't worry. Simply understand the IFFT's function is to take the frequency-domain representation of a signal, and produce the time-domain equivalent.

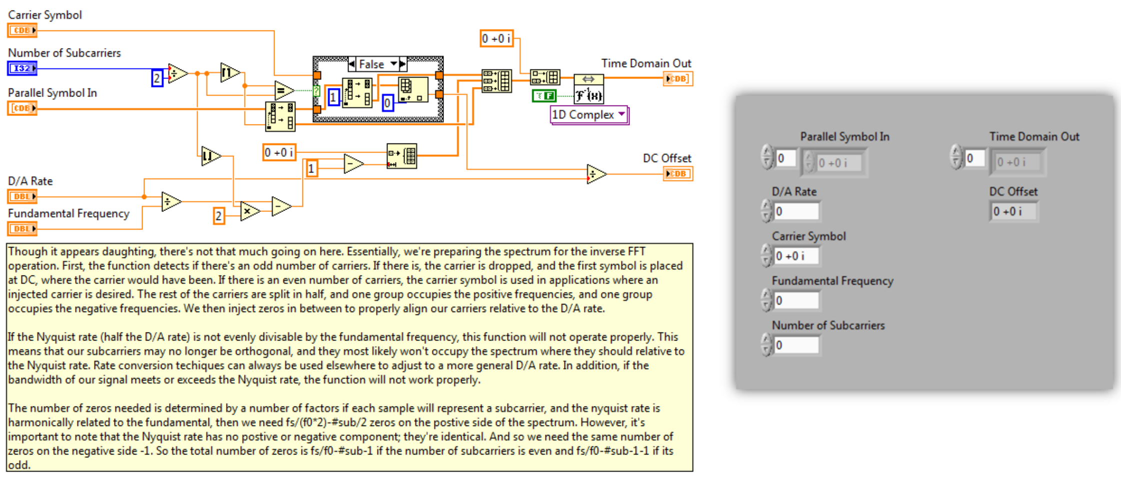
[9]http://cnx.org/content/m34127/latest/w2sym.zip
[10]This content is available online at <http://cnx.org/content/m34128/1.5/>.

So again you ask, why are we performing this step? Aren't we already in the time domain? Well, it's all rather relative. We know for OFDM the 'O' stands for Orthogonality. This attractive feature of OFDM signals makes the subcarriers insensitive to spectral overlap, much like Quadrature Multiplexing where we mix signals with cosine and sine of the same frequency. Thus as long as we modulate a variety of carriers all orthogonal to one another, we can neglect overlap of their spectra and still always recover each carrier separately without distortion. The orthogonality for OFDM comes from finding a fundamental frequency (lowest), and mixing all the other carriers with multiples of this fundamental, or harmonics. So how do we implement this?

Okay, let's look at a 32-subcarrier OFDM signal. We could modulate 32 frequencies with our 32 symbols and add them up, but this is extremely inefficient. Instead, we can start in the frequency domain, and place our symbols in adjacent samples. By doing this, because the samples are periodic, we're guarunteed to have all the frequency samples orthogonal to the first non-DC frequency. Then, since we can use Quadrature Multiplexing later on, we can simply place 15 symbols in the first 15 samples after DC, insert a bunch of zeros, and then place the last 15 symbols in the last 15 spots. So we now have 32 subcarriers modulated: +-fo, our fundamnetal, +-2fo, ... , +-14fo in the frequency domain. Essentially what we've done is build our signal from the spectrum. Because we have done this without any kind of conjugate symettry (at least not planned any), we will get a complex time-domain signal when the inverse FFT is taken. This is okay! We eventually will modulate cosine at our intermediate frequency with the real part of our baseband signal, and the imaginary part with sine.

Another way of looking at this is that we're simpling doing a parallel to serial operation. Let's get on to the actual implementation in LabVIEW.

Block Diagram Layout



Though it appears daughting, there's not that much going on here. Essentially, we're preparing the spectrum for the inverse FFT operation. First, the function detects if there's an odd number of carriers. If there is, the carrier is dropped, and the first symbol is placed at DC, where the carrier would have been. If there is an even number of carriers, the carrier symbol is used in applications where an injected carrier is desired. The rest of the carriers are split in half, and one group occupies the positive frequencies, and one group occupies the negative frequencies. We then inject zeros in between to properly align our carriers relative to the D/A rate.

If the Nyquist rate (half the D/A rate) is not evenly divisable by the fundamental frequency, this function will not operate properly. This means that our subcarriers may no longer be orthogonal, and they most likely won't occupy the spectrum where they should relative to the Nyquist rate. Rate conversion techiques can always be used elsewhere to adjust to a more general D/A rate. In addition, if the bandwidth of our signal meets or exceeds the Nyquist rate, the function will not work properly.

The number of zeros needed is determined by a number of factors if each sample will represent a subcarrier, and the nyquist rate is harmonically related to the fundamental, then we need fs/(f0*2)-#sub/2 zeros on the postive side of the spectrum. However, it's important to note that the Nyquist rate has no postive or negative component; they're identical. And so we need the same number of zeros on the negative side -1. So the total number of zeros is fs/f0-#sub-1 if the number of subcarriers is even and fs/f0-#sub-1-1 if its odd.

Figure 2.10: This is the LabVIEW block diagram for the Symbols-to-FFT sub-VI.

Above we see the actual block diagram. Though it can appear overwhelming, as long as the theory discussed above is understood, the rest is rather simple. In fact, everything we discussed above is nearly implemented in one block by the built in IFFT function! Everything we see surrounding it is put in place for user-friendliness and flexability.

In case an injected carrier for ease of receiving is desired, the user can specify a carrier symbol. The array size divided by two and rounding simply is to find out if there is an even or odd number of subcarriers. If even, the carrier specified is injected. If odd, the DC frequency is injected with the first symbol, so the lone symbol will take the place of the carrier for the sake of symettry. The reason the round toward infinity is used is simply because of the way the split array function works. The split array function simply truncates the number and cuts the array at that point, so by rounding up the odd carrier will always be in the first half.

The second half of the split is added to the end of the frequency domain. The only magic then appears in between: how many zeros do we inject and why? The answer isn't as direct, but it is a very fundamental concept. We want to conserve bandwidth, and so this is the motivation between bunching up the subcarriers as close to DC as possible. In addition, the easiest and most efficient way to make them orthogonal is to place a symbol at each sample. This way they're all related to the first non-DC frequency which is itself a fraction of the sampling (D/A) frequency.

Now, we want to ensure these relationships are maintained, as well as a few more. The user can specify the exact fundamental frequency desired, as well as the ultimate D/A rate. Two things must hold in the spectrum then: the fundamental frequency must be related to the D/A rate appropriately and in the first sample, and the Nyquist Rate of the system must be half the sampling frequency. If any of this is completely new, review the fundamentals of Digital Signal Processing (DSP). This constraint is the reason we choose the number of zeros, and why we want to choose our fundamental frequency such that sampling_freq/(2*fundamental_freq) is a whole number. Stated another way, the Nyquist Rate is evenly divisible by our fundamental. Once again, we can say the Nyquist Rate is an integer multiple of the Fundamental Frequency. This should be apparent from the above discussion. Afterall, halfway through our samples we should encounter the Nyquist Rate, the highest representable digital frequency, half the sampling rate. Every sample before this will be a fraction of this frequency. If this frequency exists at Sample 32, then our fundamental should be 1/32th of the Nyquist rate, as we will place it on a sample directly (no interpolation).

While a more generalized procedure is certainly possible, it's simply not practical in the face of transparency and computational complexity. In addition, sample rate conversion is a topic in and of itself, and any of those advanced techniques can be applied to the signal externally before transmission.

If you are still confused on how this works, check out the instructional and example video below or email the author for more questions. The sub-VI is available below for download.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/PFeAlHbhkX8&hl=en&fs=1&rel=0>

Figure 2.11: This is the instructional video for constructing the Symbols-to-FFT prefix sub-VI.

Example Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/-TmALixLjF4&hl=en&fs=1&rel=0>

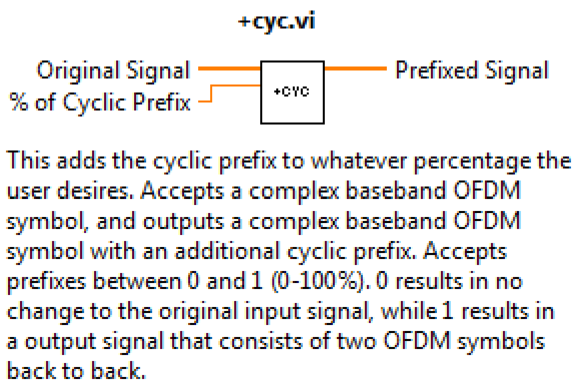
Figure 2.12: This is the example video for using the Symbols-to-FFT sub-VI.

Download This LabVIEW sub-VI[11]

[11]http://cnx.org/content/m34128/latest/sym2fft.zip

2.2.4 Add Cyclic Prefix (Transmitter) (LabVIEW)[12]

Input/Outputs and Help



This adds the cyclic prefix to whatever percentage the user desires. Accepts a complex baseband OFDM symbol, and outputs a complex baseband OFDM symbol with an additional cyclic prefix. Accepts prefixes between 0 and 1 (0-100%). 0 results in no change to the original input signal, while 1 results in a output signal that consists of two OFDM symbols back to back.

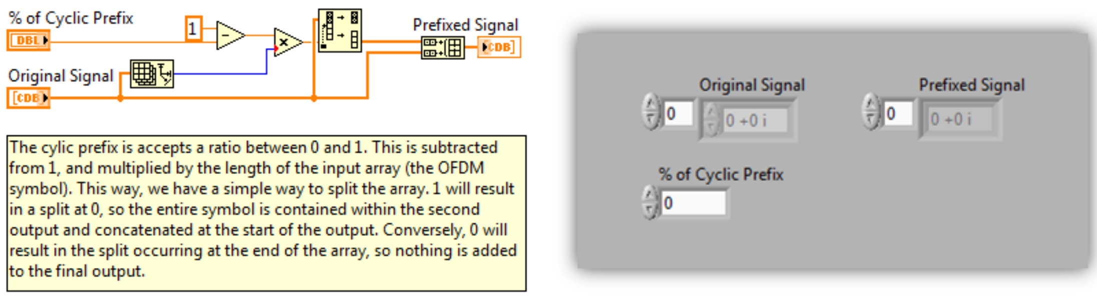
Figure 2.13: This is the LabVIEW help and block description for the cyclic prefix sub-VI.

The Cyclic Prefix sub-VI in the OFDM transmitter is shown above in Figure 1. The theory behind the cyclic prefix is rather intuitive. Leading up to this module, we have one complete period of a complex baseband OFDM symbol. However, depending on the channel, we may have significant multipath fading. In order to combat channels with delays, we need a guard interval. This is essentially an interval in the transmission that we will eventually throw out. For more complete treatment on multipath channel models and why guard intervals are effective, Krishna Sankar does a terrific job in his blog entry[13] .

The procedure is very simple. Some percentage of the end of the signal is copied and added to the front. This doesn't create any discontinuities because the next point after the end of the signal is the beginning of the signal since we have exactly one period. Thus any portion of the end of the signal we copy to the front of the signal will add without discontinuity. If this is still difficult to see, imagine copying and pasting a full period of a sine wave to the start, so we now have two periods of that sine wave. Now imagine deleting 1/4 of the points at the beginning of the new signal. It's easy to see that nothing unorthodox is going on here, and you've effectively created a cyclic prefix of 0.5 or 50%. Now it's easy to generalize this to any arbitrary signal, as long as we have exactly one period.

[12]This content is available online at <http://cnx.org/content/m34119/1.5/>.
[13]http://www.dsplog.com/2008/02/17/cylcic-prefix-in-orthogonal-frequency-division-multiplexing/

Block Diagram Layout



The cylic prefix is accepts a ratio between 0 and 1. This is subtracted from 1, and multiplied by the length of the input array (the OFDM symbol). This way, we have a simple way to split the array. 1 will result in a split at 0, so the entire symbol is contained within the second output and concatenated at the start of the output. Conversely, 0 will result in the split occurring at the end of the array, so nothing is added to the final output.

Figure 2.14: This is the LabVIEW block diagram for the cyclic prefix sub-VI.

Here, Figure 2 shows a LabVIEW implementation of what we just described. It is available for download below. The only subtleties here mainly deal with user friendly-ness. The module accepts a ratio between 0 and 1 representing a percentage between 0 and 100%. We then take the complement of this ratio in order to properly split the array. So our ratio is created by 1-(perc/100). This is because LabVIEW's built in function split array takes as an input the point in the array at which we wish to split. Thus if we want 75% prefix, since we work from the end of the signal and move towards the start, we want to break at the 25% point of the signal and take the second half. This ratio is thus multiplied by the total signal length and the break point is calculated.

The portion that is extracted is then concatenated to the original input signal, and our prefixed signal is complete. One sticking point is to ensure that when the "Build Array" function is invoked, that you right-click the block and choose "Concatenate Inputs" so that the arrays are properly joined. For all general questions, check out the instructional video below in Figure 3, the example use video in Figure 4, or email the author for more information.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/Zykhdaw_YkI&hl=en&fs=1&rel=0>

Figure 2.15: This is the instructional video for constructing the cyclic prefix sub-VI.

Example Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/BzN5nWRsslA&hl=en&fs=1&rel=0>

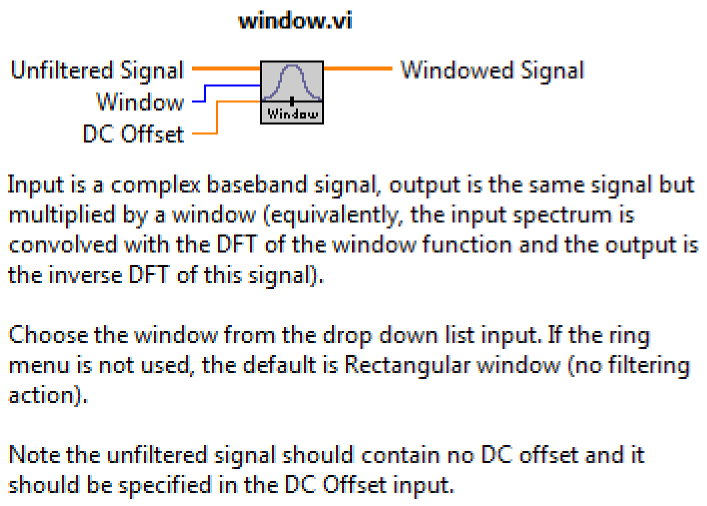
Figure 2.16: This is the example video for using the cyclic prefix sub-VI.

Download This LabVIEW sub-VI[14]

[14]http://cnx.org/content/m34119/latest/cyc.zip

2.2.5 Windowing (Transmitter) (LabVIEW)[15]

Input/Outputs and Help



Input is a complex baseband signal, output is the same signal but multiplied by a window (equivalently, the input spectrum is convolved with the DFT of the window function and the output is the inverse DFT of this signal).

Choose the window from the drop down list input. If the ring menu is not used, the default is Rectangular window (no filtering action).

Note the unfiltered signal should contain no DC offset and it should be specified in the DC Offset input.

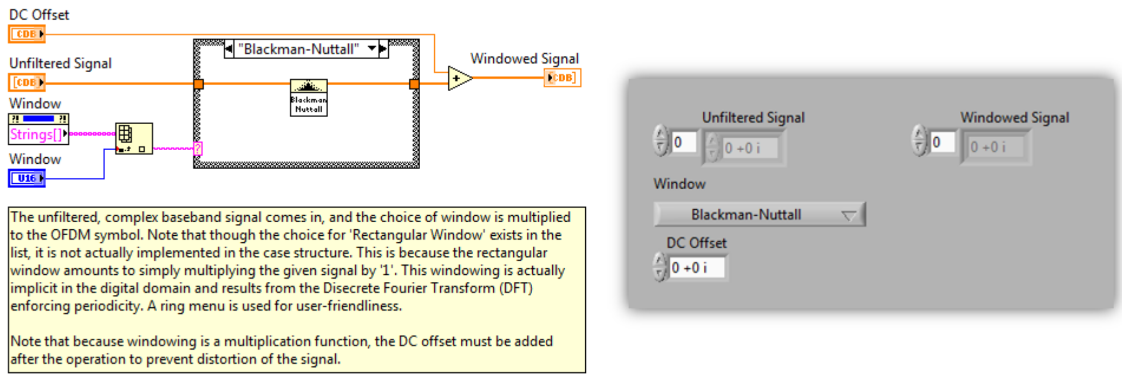
Figure 2.17: This is the LabVIEW help and block description for the Window Sub- VI.

The above figure shows the sub-VI snapshot of the rather simple module that windows the outgoing signal. This VI can be imperative in certain systems. The reason it becomes critical is the inevitable discontinuity that will arise between OFDM symbols.

If this is unclear, make sure you understand just how OFDM symbols are synthesized. The signals are built from symbol placement in the frequency domain at harmonics of the fundamental. As a result, there is no time-domain cohesiveness between symbols (or at least it occurs rarely). This is the same property (essentially) that plagues OFDM systems in the form of peak-to-average power ratios.

[15]This content is available online at <http://cnx.org/content/m34124/1.5/>.

Block Diagram Layout



The unfiltered, complex baseband signal comes in, and the choice of window is multiplied to the OFDM symbol. Note that though the choice for 'Rectangular Window' exists in the list, it is not actually implemented in the case structure. This is because the rectangular window amounts to simply multiplying the given signal by '1'. This windowing is actually implicit in the digital domain and results from the Disecrete Fourier Transform (DFT) enforcing periodicity. A ring menu is used for user-friendliness.

Note that because windowing is a multiplication function, the DC offset must be added after the operation to prevent distortion of the signal.

Figure 2.18: This is the LabVIEW block diagram for the Window sub-VI.

Pictured above is the LabVIEW block diagram implementation of the windowing process. Thankfully, almost all prominent window functions are built into LabVIEW. The windowing process consists of a simple multiplication in the time-domain. Those familiar with Fourier decomposition will recognize this corresponds to convolving in the frequency domain. Usually, practical windowing involves emphasizing the middle of the time domain signal, and deemphasizing the beginning and the end of the signal.

The reason for this deemphasis stems from the fundamental (but covert) function the Discrete Fourier Transform (DFT) performs. While subtle, it's important to note that by taking the DFT of a signal, one is enforcing periodicty of that signal. That is, we are transforming the signal into the frequency domain under the assumption that the input repeats in the time-domain indefinitely. As a result, the un-windowed signal (or equivalently, rectangularly windowed signals) present serious flaws from this poor assumption. This is especially manifested in our situation in which discontinuities are almost guarunteed to occur. Therefore, deemphasizing the end points, and consequently the discontinuities, proves fruitful.

The only slight of hand in the LabVIEW implementation is the window selection method. After creating a case structure, simply naming the various windows with strings and inserting them suffices. However, in the interest of user-friendliness, an effective interface is needed. By creating a menu-ring, the options easily present themselves, and a simple array index operation allows us to access the case structure with ease. For more information, check out the instructional tutorial below, the example video in Figure 4, or email the author.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/BDdNJam9xOU&hl=en&fs=1&rel=0>

Figure 2.19: This is the instructional video for constructing the windowing sub-VI.

Example Video

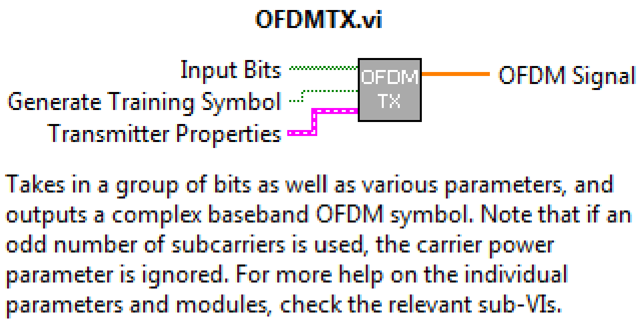
This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/QfGp69bniVo&hl=en&fs=1&rel=0>

Figure 2.20: This is the example video for using the windowing sub-VI.

Download This LabVIEW sub-VI[16]

2.2.6 OFDM Symbol Generator (Transmitter) (LabVIEW)[17]

Input/Outputs and Help



Takes in a group of bits as well as various parameters, and outputs a complex baseband OFDM symbol. Note that if an odd number of subcarriers is used, the carrier power parameter is ignored. For more help on the individual parameters and modules, check the relevant sub-VIs.

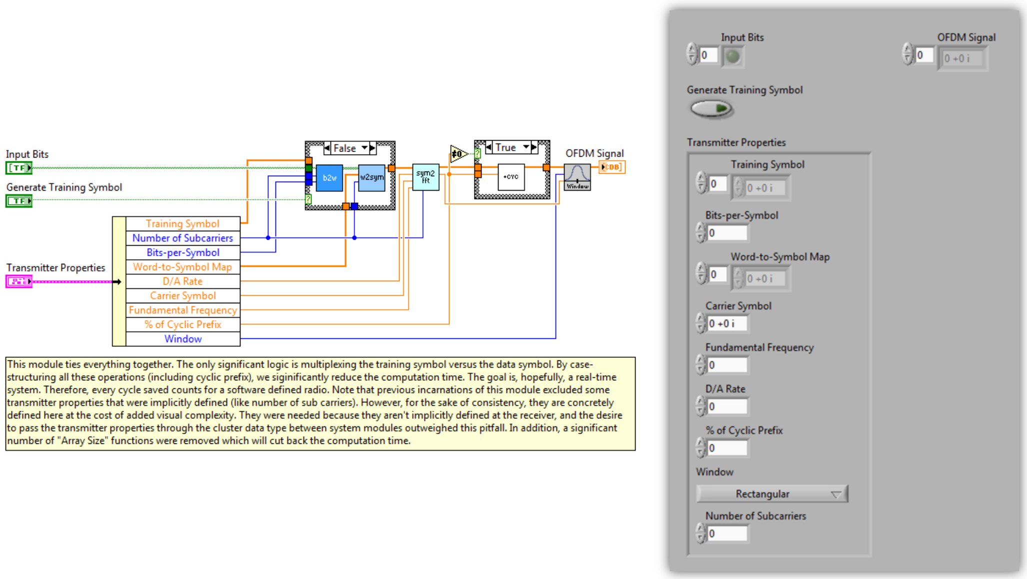
Figure 2.21: This is the LabVIEW help and block description for the OFDM Symbol Generator Sub-VI.

Putting together all of the previously built sub-VI's into a sub-VI itself, we get a convienent block (shown above in Figure 1) to drop into any software communications template in order to complete the system. By specifying all the input data, this block will output a signal OFDM time-domain signal ready to be quadrature multiplexed out to IF and actually transmitted.

Again, though some may find this module unecessary (and pedantically speaking they would be correct), modularity and transparency goes a long way in system integration. Almost always, the benefits outweigh the minimal time needed to develop such modules.

[16]http://cnx.org/content/m34124/latest/window.zip
[17]This content is available online at <http://cnx.org/content/m34129/1.5/>.

Block Diagram Layout



This module ties everything together. The only significant logic is multiplexing the training symbol versus the data symbol. By case-structuring all these operations (including cyclic prefix), we siginificantly reduce the computation time. The goal is, hopefully, a real-time system. Therefore, every cycle saved counts for a software defined radio. Note that previous incarnations of this module excluded some transmitter properties that were implicitly defined (like number of sub carriers). However, for the sake of consistency, they are concretely defined here at the cost of added visual complexity. They were needed because they aren't implicitly defined at the receiver, and the desire to pass the transmitter properties through the cluster data type between system modules outweighed this pitfall. In addition, a significant number of "Array Size" functions were removed which will cut back the computation time.

Figure 2.22: This is the LabVIEW block diagram for the OFDM Symbol Generator sub-VI.

There is little to discuss here, and the really only sticking point is the use of the case structures and clusters. One could have the blocks outside the case structures and feed their outputs through the case structures. However, by placing the functions in the case structure, we decide if we actually call the function or not, instead of using the output signal. This way we reduce unneeded computational time. The cluster data type is used for visual clarity, and ease of transferring transmitter properties to other systems (like the receiver).

If you are still confused on how any of this works, check out the instructional video below, the all software test module, or email the author for more questions. The sub-VI (as well as all VIs it uses) is available below for download.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/QWzYEy9gshs&hl=en&fs=1&rel=0>

Figure 2.23: This is the instructional video for constructing the OFDM Symbol Generator sub-VI.

Download This LabVIEW sub-VI[18]
NOTE: This zip file also contains all the sub-VI's shown in this module that this sub-VI uses.

[18]http://cnx.org/content/m34129/latest/ofdmtx.zip

Chapter 3

Receiver

3.1 MATLAB

3.1.1 Time Synchronizaton (Receiver) (MATLAB)[1]

```
%% Time Synchronization (RECEIVER)
% ------------------------------------------------------------------------
% Description: This implements Schmidl and Cox's symbol frame timing
%              synchronization algorithm. The idea is that by zeroing out
%              the odd frequencies for the training symbol, we produce an
%              OFDM symbol that has two periods within the normal OFDM
%              symbol frame, whereas a normal symbol has only one period.
%              By sliding a window in time and multiplying sample by
%              sample two successive windows of length half of the normal
%              symbol and one of them conjugated, the magnitude peak
%              should hit '1' and represent the exact start of the frame
%              for the training symbol. Thus it as well as the data symbol
%              can be extracted
%
% Inputs: signal - Input waveform, sampled from LABVIEW
%         size_of_fft - Size of fft for symbol size
% Outputs: index - Index of start of timing frame
%           agc - Automatic gain control factor

function [index agc] = tsync(signal, size_of_fft)
%%
    signal_size = size(signal,2); % Signal size
    L = size_of_fft/2; % Length of sliding window
    slide_length = signal_size-size_of_fft+1; % Total window slide length
    P = zeros(1,slide_length); % Initilize the arrays
    R = zeros(1,slide_length);

    for n=1:slide_length
        for m=1:L
            P(n) = P(n) + conj(signal(n+m-1))*signal(n+m+L-1);
```

[1]This content is available online at <http://cnx.org/content/m33829/1.2/>.

```
            % Conjugate pair
            R(n) = R(n) + abs(signal(n+m+L-1))^2; % Normalizing factor
        end
    end
    M = (abs(P).^2);%./(R.^2);
    % This is the actual timing metric used. Note the algorithm calls for
    % the normalization factor. However, due to peak-to-average power
    % issues, we experienced issues with the algorithm choosing false peaks
    % that were greater than 1. After much experimenting, removing the
    % normalization factor and boosting the signal power of just the
    % odd-channels in the training symbols made this algorithm robust.
    for n=1:slide_length
        [value index] = max(M); % Find peak
        if((index+2*size_of_fft-1) > signal_size)
            M = M(1:index-1); % If peak is near the end, and the symbols cut off, back up and pick anot
        else
            break
        end
    end

    agc = R(index); % Output normalizing factor for automatic gain control (optional)
end
```

3.1.2 Frequency Synchronization (Receiver) (MATLAB)[2]

```
%% Frequency Synchronization (RECEIVER)
%  ------------------------------------------------------------------------
%  Description: This module attempts to correct the frequency/phase offset
%               in the recovered signal by estimating it as a static phase
%               offset. Though there may be a frequency offset or even a
%               frequency drift, for BPSK, estimating this as a fixed phase
%               offset is sufficient in recovering the bits. More often
%               than not this corrects the recovered bits when they're
%               inverted (estimated as a pi phase offset). This uses the
%               same metric as the timing recovery to find the phase.
%
%  Inputs: train - Received training symbol
%          train_expected - Expected training symbol via a priori knowledge
%  Outputs: phi - Estimated phase offset

function phi = fsync(train, train_expected)
%%
     L = size(train,2); % Size of window for algorithm
     P = 0; % Initialize the metric

     for n=1:L
         P = P + conj(train(n))*train_expected(n);
         % Sample by sample multiplication of the complex conjugate of the
```

[2] This content is available online at <http://cnx.org/content/m33848/1.2/>.

```
        % received training symbol with the expected training symbol. If
        % both of them are in phase, the imaginary components would
        % annhiliate each other. However, if there's a phase offset
        % between the two, it will be represented in the angle of the
        % resulting product.
     end
    phi = angle(P);
end
```

3.1.3 FFT-to-Symbols (Receiver) (MATLAB)[3]

```
    %% FFT to Symbols (RECEIVER)
%  ---------------------------------------------------------------------------
%  Description: Takes a received OFDM signal that has been demodulated to
%               baseband and corrected for frequency/phase offset, and
%               extracts the complex envelope of each carrier. This
%               converts the time domain signal into a descrete set of
%               symbols that can be mapped to binary words.
%
%  Inputs: re - Real channel of OFDM symbol
%          im - Imaginary channel of OFDM symbol
%          num_subcarriers - Number of subcarriers used in OFDM symbol
%  Outputs: symbols - Extracted complex envelopes of each subcarrier

function symbols = fft2sym(re, im, num_subcarriers)
%%

    size_of_fft = size(re,2); % Implicit size of fft
    output = fft(re+j*im); % Time domain signal was generated using ifft()
    symbols = [ output(2:1+num_subcarriers/2) output(size_of_fft+1-num_subcarriers/2:size_of_fft) ];
    % Save only the FFT samples that contain complex symbols as specified
    % by the number of subcarriers.
end
```

3.1.4 Symbols-to-Words (Receiver) (MATLAB)[4]

```
%% Symbols to Words (RECEIVER)
%  ---------------------------------------------------------------------------
%  Description: Converts the received symbols from the OFDM symbol spectrum
%               into binary words. It does this by correlating the received
%               symbol with the entire symbol map through a minimization of
%               the absolute value of the difference with each. Note that
%               if one uses multiple constellation points in the same
%               quandrant of the complex plane, it may be necessary to add
%               additional logic here, or ensure the channel effect is
%               removed by equalization.
```

[3]This content is available online at <http://cnx.org/content/m33849/1.2/>.
[4]This content is available online at <http://cnx.org/content/m33850/1.2/>.

```
%
%  Inputs: symbols - Symbols extracted from the OFDM symbol
%          word_to_symbol_map - The complex symbols indexed by the binary
%                               word they represent.
%  Outputs: words - Closest matched binary words after comparison

function words = sym2w(symbols, word_to_symbol_map)
%%
    num_subcarriers = size(symbols,2); % Implicit number of sub-carriers
    bits_per_symbol = log2(size(word_to_symbol_map,2));
    % Implicit number of bits per symbol to represent each word
    words = zeros(num_subcarriers, bits_per_symbol); % Initialized array
    for n=1:num_subcarriers
        [v i] = min(abs(word_to_symbol_map-symbols(n)));
        % Subtract each symbol with every symbol in the map, and find the
        % minimum absolute value. This is a great estimate for finding the
        % received word.
        word = dec2bin(i-1,bits_per_symbol); % Convert index to binary string
        for m=1:bits_per_symbol
            words(n,m) = str2num(word(m)); % Convert string to number
        end
    end
end
```

3.1.5 Words-to-Bits (Receiver) (MATLAB)[5]

```
%% Words to Bits (RECEIVER)
%  ------------------------------------------------------------------------
%  Description: This module simply takes the demodulated words and expands
%               them into a bit stream for ease of comparison.
%
%  Inputs: words - Group of bits
%  Outputs: bits - Bit stream

function bits = w2b(words)
%%
    num_subcarriers = size(words,1); % Implicit number of subcarriers
    bits_per_symbol = size(words,2); % Implicit bits per symbol
    total_bits = bits_per_symbol*num_subcarriers; % Total number of bits
    bits = zeros(1,total_bits);

    for n=1:num_subcarriers
        for m=1:bits_per_symbol
            bits((n-1)*bits_per_symbol+m) = words(n,m); % Separate one by one
        end
    end
end
```

[5]This content is available online at <http://cnx.org/content/m33851/1.2/>.

3.1.6 OFDM Symbol Decoder (Receiver) (MATLAB)[6]

```
%% Demodulate Baseband OFDM Signal (RECEIVER)
% ---------------------------------------------------------------------
% Description: This is the receiver module in its entirity. It combines
%              all of the sub-processes and parameters to ultimately take
%              a sequency of training symbol and valid OFDM symbol, and
%              output the modulated bits.
%
% Inputs: word_to_symbol_map - The complex symbols indexed by the binary
%                              word they represent.
%         num_subcarriers - The number of subcarriers in the symbol
%         size_of_fft - Size of FFT used for correct indexing/timing
%         train_pn - Pseudo-noise used in transmitted training symbol (a
%                    priori knowledge)
% Outputs: bits - Demodulated bits

function bits = ofdm_dem(word_to_symbol_map, num_subcarriers, size_of_fft, train_pn)
%%
% Calculated the expected training symbol from the given pseudo-noise for
% adequate phase recovery.
train_expected = zeros(1,size_of_fft);
train_expected(3) = train_pn(1);
train_expected(size_of_fft-1) = train_pn(1+size(train_pn,2)/2);
train_expected = ifft(4*train_expected);
z = get_input(); % Parse input data
z = z'; % Convert from column vector to row vector to accomidate sub-processes
% Invoke timing algorithm to retrieve sample that training symbol starts on
[index value] = tsync(z,size_of_fft);
train = z(index:index+size_of_fft-1); % Cut out training symbol
phi = fsync(train, train_expected); % Estimate phase offset
% Cut out OFDM data symbol and correct for phase offset
z = z(index+size_of_fft:index+2*size_of_fft-1).*exp(j*phi);

% Separate the data symbol into two channels and take FFT for analysis
% below
re = real(z);
im = imag(z);
Z = fft(z);

symbols = fft2sym(re,im,num_subcarriers); % Take FFT and retrieve symbols
words = sym2w(symbols,word_to_symbol_map); % Find closest match and map words to symbols
bits = fliplr(w2b(words)); % Decompress words into a series of bits for recovery.

% Output the resulting data for debugging and analysis
subplot(5,2,1); plot(real(train)); title('Re[Training Symbol]');
subplot(5,2,2); plot(imag(train)); title('Im[Training Symbol]');
subplot(5,2,3); plot(re); title('Re[OFDM Data Symbol]');
subplot(5,2,4); plot(im); title('Im[OFDM Data Symbol]');
```

[6]This content is available online at <http://cnx.org/content/m33852/1.2/>.

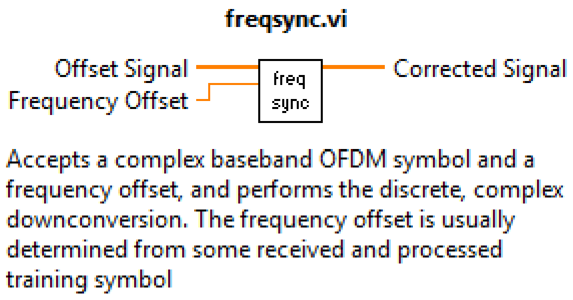
```
subplot(5,2,[5 6]); plot(abs(Z)); title('Mag[OFDM Data Symbol Spectrum]');
subplot(5,2,[7 8]); stem(real(Z)); title('Re[OFDM Data Symbol Spectrum]');
subplot(5,2,[9 10]); stem(imag(Z)); title('Im[OFDM Data Symbol Spectrum]');

end
```

3.2 LabVIEW

3.2.1 Frequency Synchronization (Receiver) (LabVIEW)[7]

Input/Outputs and Help

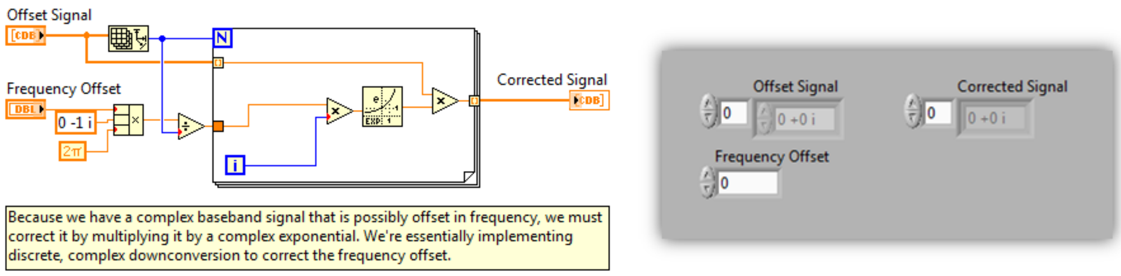


Accepts a complex baseband OFDM symbol and a frequency offset, and performs the discrete, complex downconversion. The frequency offset is usually determined from some received and processed training symbol

Figure 3.1: This is the LabVIEW help and block description for the Frequency Synchronization sub-VI.

The Frequency Synchronization sub-VI in the OFDM transmitter is shown above in Figure 1. The motivation behind this sub-VI arises because we assume in general we have a complex baseband OFDM symbol. As a result, any frequency offset from noncoherence up until this point must be corrected by a complex downconversion. Of course, maintaining coherence is crucial in preserving the orthogonality of the subcarriers and ultimately in distortionless recovery of the data.

Block Diagram Layout



Because we have a complex baseband signal that is possibly offset in frequency, we must correct it by multiplying it by a complex exponential. We're essentially implementing discrete, complex downconversion to correct the frequency offset.

Figure 3.2: This is the LabVIEW block diagram for the Frequency Synchronization sub-VI.

[7]This content is available online at <http://cnx.org/content/m34347/1.1/>.

Figure 2 shows a LabVIEW implementation of this downconversion and is available for download below. The only potentially unseen function for the new LabVIEW user is the compound arithmetic function. This can be found in the numeric menu. Once placed in your VI, you must right-click the block and choose the arithmetic operation you desire (multiplication in our case). Note also that indexing is enabled for ease of implementation.

For all general questions, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

This media object is a Flash object. Please view or download it at <http://www.youtube.com/v/FBN7BC6CA0A&hl=en&fs=1&rel=0>

Figure 3.3: This is the instructional video for constructing the Frequency Synchronization sub-VI.

Example Video

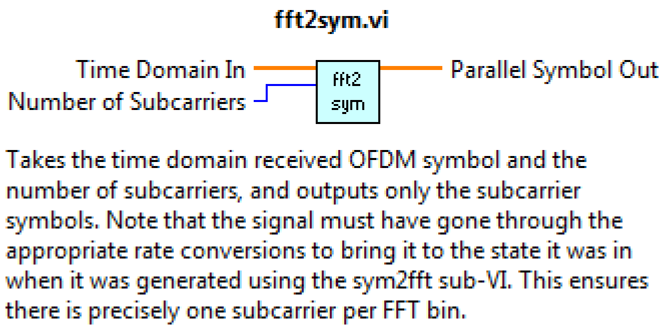
This media object is a Flash object. Please view or download it at <http://www.youtube.com/v/haxRSUTu9Gg&hl=en&fs=1&rel=0>

Figure 3.4: This is the example video for using the Frequency Synchronization sub-VI.

Download This LabVIEW sub-VI[8]

3.2.2 FFT-to-Symbols (Receiver) (LabVIEW)[9]

Input/Outputs and Help



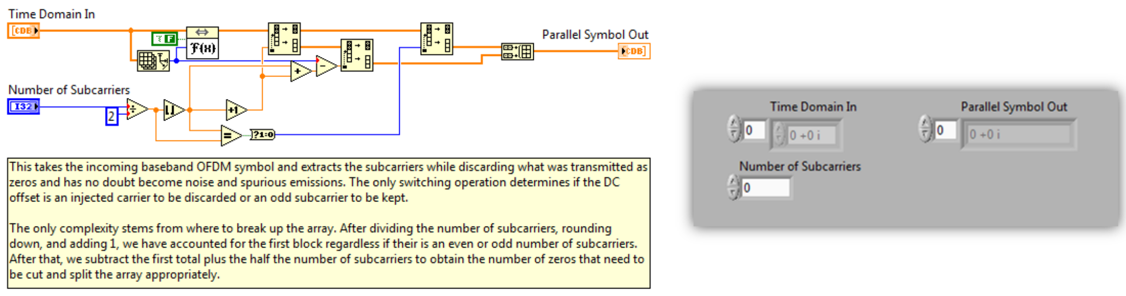
Takes the time domain received OFDM symbol and the number of subcarriers, and outputs only the subcarrier symbols. Note that the signal must have gone through the appropriate rate conversions to bring it to the state it was in when it was generated using the sym2fft sub-VI. This ensures there is precisely one subcarrier per FFT bin.

Figure 3.5: This is the LabVIEW help and block description for the FFT-to-Symbols sub-VI.

[8]http://cnx.org/content/m34347/latest/freqsync.zip
[9]This content is available online at <http://cnx.org/content/m34351/1.1/>.

Shown above is the sub-VI to be discussed next. Once we receive our time domain signal, we need to recover the subcarriers and ultimately the data they contain. We took the IFFT at the transmitter, and so it is a simple matter of taking the FFT and removing the padded zeros.

Block Diagram Layout



This takes the incoming baseband OFDM symbol and extracts the subcarriers while discarding what was transmitted as zeros and has no doubt become noise and spurious emissions. The only switching operation determines if the DC offset is an injected carrier to be discarded or an odd subcarrier to be kept.

The only complexity stems from where to break up the array. After dividing the number of subcarriers, rounding down, and adding 1, we have accounted for the first block regardless if their is an even or odd number of subcarriers. After that, we subtract the first total plus the half the number of subcarriers to obtain the number of zeros that need to be cut and split the array appropriately.

Figure 3.6: This is the LabVIEW block diagram for the FFT-to-Symbols sub-VI.

Figure 2 shows the block diagram of our implementation available for download below. After taking the FFT, we simply determine how much of the spectrum should be data, and remove it from the rest of the array. While we transmitted a specific number of zeros in between the positive and negative frequencies, they will no doubt take on some small non-zero value after realistic travel through a non-software medium.

The only real logic is in dividing the number of subcarriers by 2 to determine if we have an even or odd number of subcarriers. If even, we can throw away the first data point as it is simply an injected carrier (or nothing). However, if odd, it instead contains a data point for the odd subcarrier and must be preserved in the output. For additional information, check out the tutorial video in Figure 3, the example video in Figure 4, or email the author with any questions.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/6uZ5N1cR7CA&hl=en&fs=1&rel=0>

Figure 3.7: This is the instructional video for constructing the FFT-to-Symbols sub-VI.

Example Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/1GJqHmOhk28&hl=en&fs=1&rel=0>

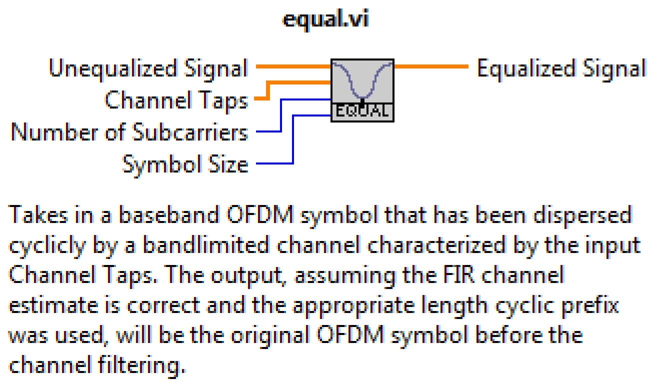
Figure 3.8: This is the example video for using the FFT-to-Symbols sub-VI.

Download This LabVIEW sub-VI[10]

[10]http://cnx.org/content/m34351/latest/fft2sym.zip

3.2.3 Equalization (Receiver) (LabVIEW)[11]

Input/Outputs and Help



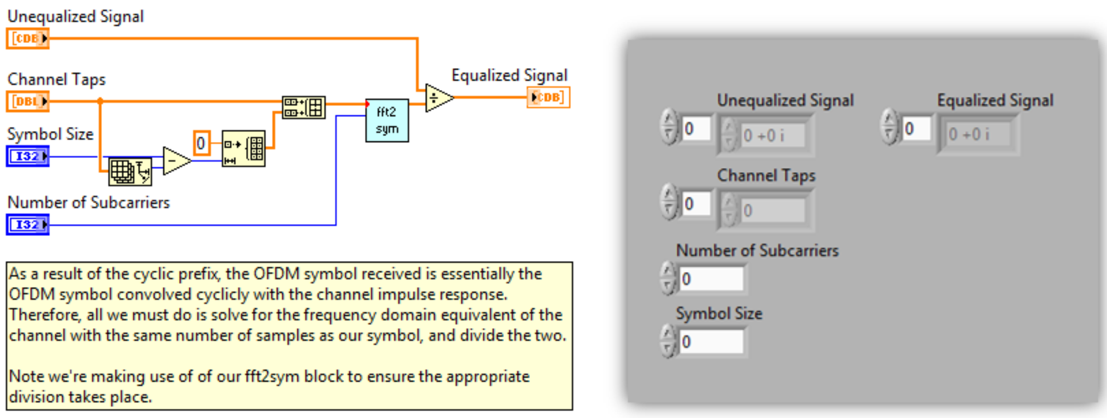
Takes in a baseband OFDM symbol that has been dispersed cyclicly by a bandlimited channel characterized by the input Channel Taps. The output, assuming the FIR channel estimate is correct and the appropriate length cyclic prefix was used, will be the original OFDM symbol before the channel filtering.

Figure 3.9: This is the LabVIEW help and block description for the Equalization sub-VI.

In Figure 1 we show the help dialog for the Equalization sub-VI to be implemented below. The motivation is that nearly every practical channel is bandlimited, and so our received signal will ultimately be the transmitted signal filtered by the dispersive channel. The resulting intersymbol interference (ISI) is the reason why a cyclic prefix, or guard interval, is required.

However, even if the cyclic prefix takes the brunt of the intersymbol interference, our received OFDM symbol has still experienced cyclic convolution with the channel impulse response. Thankfully, this amounts to normal division in the frequency domain if we can estimate the channel taps.

Block Diagram Layout



As a result of the cyclic prefix, the OFDM symbol received is essentially the OFDM symbol convolved cyclicly with the channel impulse response. Therefore, all we must do is solve for the frequency domain equivalent of the channel with the same number of samples as our symbol, and divide the two.

Note we're making use of of our fft2sym block to ensure the appropriate division takes place.

Figure 3.10: This is the LabVIEW block diagram for the Equalization sub-VI.

[11]This content is available online at <http://cnx.org/content/m34348/1.1/>.

The implementation to be discussed in this module can be seen above in Figure 2. It's quite simple for those familiar with basic DSP. If we know the time domain impulse response of the channel (or can estimate it), we can simply take the FFT to obtain the frequency response. After removal of the cyclic prefix, it's as if the channel frequency response was simply multiplied by our transmitted OFDM symbol with no zero-padding for either digital spectrum. Therefore, all we must do is divide to undo this unwanted distortion.

Note that we have taken advantage of our fft2sym sub-VI implemented previously[12] . For all general questions, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/rIRg2C5w31U&hl=en&fs=1&rel=0>

Figure 3.11: This is the instructional video for constructing the Equalization sub-VI.

Example Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/-vtb-fBAWhQ&hl=en&fs=1&rel=0>

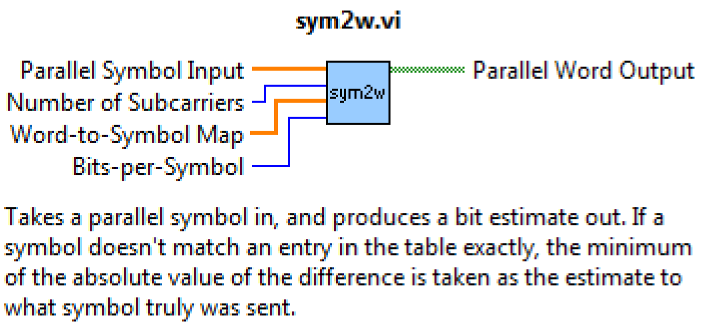
Figure 3.12: This is the example video for using the Equalization sub-VI.

Download This LabVIEW sub-VI[13]

[12]http://cnx.org/content/m34348/latest/
[13]http://cnx.org/content/m34348/latest/equal.zip

3.2.4 Symbols-to-Words (Receiver) (LabVIEW)[14]

Input/Outputs and Help



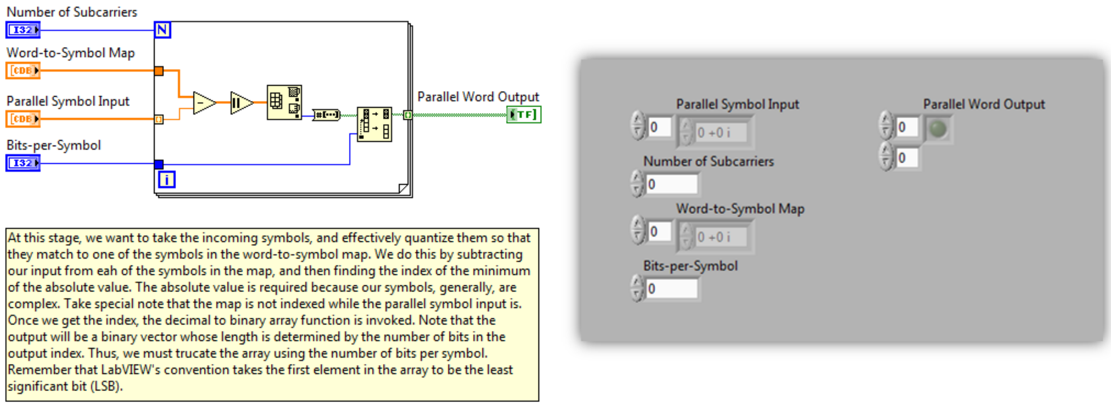
Takes a parallel symbol in, and produces a bit estimate out. If a symbol doesn't match an entry in the table exactly, the minimum of the absolute value of the difference is taken as the estimate to what symbol truly was sent.

Figure 3.13: This is the LabVIEW help and block description for the Symbols-to-Words sub-VI.

Above is the sub-VI we intend to build in this module. This is the first module where we've attempted to address the stochastic nature of our received data. We'll take each received symbol and compare it with our word-to-symbol map. The closest match is determined to be the most likely transmitted symbol.

The reason the incoming signals may not perfectly match our transmitted symbols is because realistically, computational artifacts from the FFT operation, noise, dispersive channels, quantization, and imperfect coherent carrier recovery are just a handful of the potential sources that can cause our data to change from the transmitter to the receiver.

Block Diagram Layout



At this stage, we want to take the incoming symbols, and effectively quantize them so that they match to one of the symbols in the word-to-symbol map. We do this by subtracting our input from eah of the symbols in the map, and then finding the index of the minimum of the absolute value. The absolute value is required because our symbols, generally, are complex. Take special note that the map is not indexed while the parallel symbol input is. Once we get the index, the decimal to binary array function is invoked. Note that the output will be a binary vector whose length is determined by the number of bits in the output index. Thus, we must trucate the array using the number of bits per symbol. Remember that LabVIEW's convention takes the first element in the array to be the least significant bit (LSB).

Figure 3.14: This is the LabVIEW block diagram for the Symbols-to-Words sub-VI.

Above, Figure 2 shows a LabVIEW implementation of our slicer. As alluded to, we essentially compute minimum Euclidean distance between all allowed values of our symbols (from our known map) and our

[14]This content is available online at <http://cnx.org/content/m34345/1.1/>.

received data one subcarrier at a time. It's important to note indexing is enabled for our incoming data to perform our operation one data carrier at a time, but the word-to-symbol map is not, as we want to compare each incoming data symbol with all symbols in the map.

Once we take advantage of LabVIEW's max/min function, we desire only the index as this corresponds to the decimal equivalent of our binary word. Remember LabVIEW's convention that the first element in the output array corresponds to the least significant bit. Because the number of bits varies depending on the size of the map, the truncation shown at the end is necessary to remove excess leading zeros. For all general questions, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/dpdp_u6lgOA&hl=en&fs=1&rel=0>

Figure 3.15: This is the instructional video for constructing the Symbols-to-Words sub-VI.

Example Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/jSvw40Sp1C8&hl=en&fs=1&rel=0>

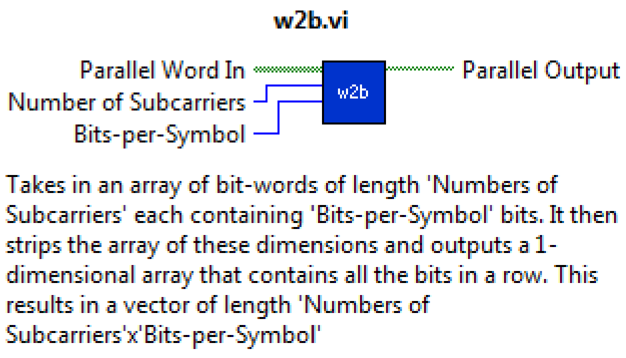
Figure 3.16: This is the example video for using the Symbols-to-Words sub-VI.

Download This LabVIEW sub-VI[15]

[15]http://cnx.org/content/m34345/latest/sym2w.zip

3.2.5 Words-to-Bits (Receiver) (LabVIEW)[16]

Input/Outputs and Help

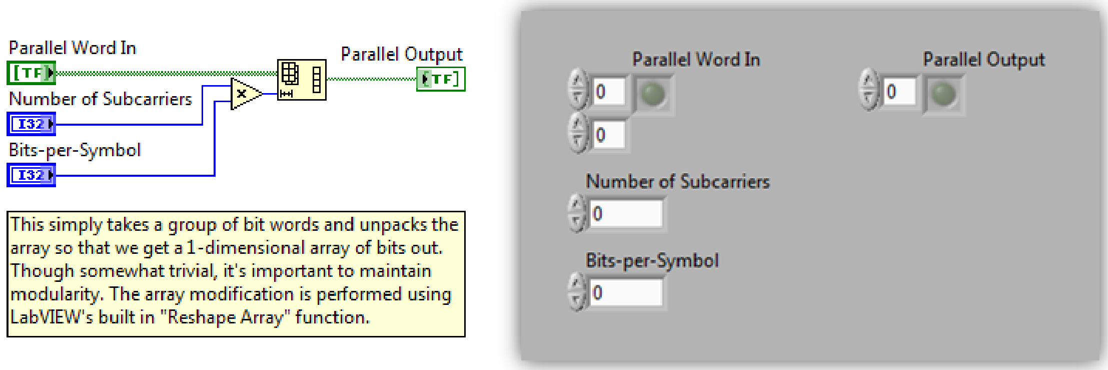


Takes in an array of bit-words of length 'Numbers of Subcarriers' each containing 'Bits-per-Symbol' bits. It then strips the array of these dimensions and outputs a 1-dimensional array that contains all the bits in a row. This results in a vector of length 'Numbers of Subcarriers'x'Bits-per-Symbol'

Figure 3.17: This is the LabVIEW help and block description for the Word-to-Bits sub-VI.

We come again to a seemingly undeserving sub-VI shown above. While it may not warrant a sub-VI at present, future implementations will benefit from it's separation from other sub-VIs. This way, upgrading and changing individual parts will have minimal impact on the adjacent subsystems.

Block Diagram Layout



This simply takes a group of bit words and unpacks the array so that we get a 1-dimensional array of bits out. Though somewhat trivial, it's important to maintain modularity. The array modification is performed using LabVIEW's built in "Reshape Array" function.

Figure 3.18: This is the LabVIEW block diagram for the Words-to-Bits sub-VI.

The above figure is our simple implementation as discussed above. All we're doing is taking advantage of LabVIEW's reshape array function. We're reshaping our two dimensional array defined by the input integers to one continuous array comprising one dimension.

Note that optimal implementations at lower levels could be achieved implicitly by how the memory is mapped. Therefore this module is simply a convenient educational tool for transparency in observing the

[16]This content is available online at <http://cnx.org/content/m34344/1.1/>.

overall OFDM transceiver hierarchy. For any inquiries, check out the instructional video below in Figure 3, the example use video below in Figure 4, or email the author for more information.

Instructional Video

This media object is a Flash object. Please view or download it at <http://www.youtube.com/v/q4_1VWejcgI&hl=en&fs=1&rel=0>

Figure 3.19: This is the instructional video for constructing the Words-to-Bits sub-VI.

Example Video

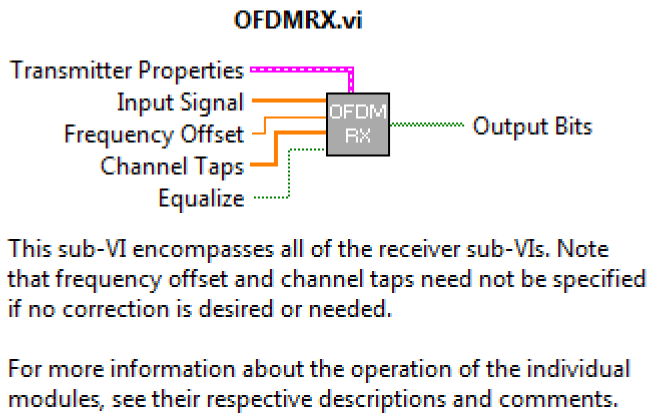
This media object is a Flash object. Please view or download it at <http://www.youtube.com/v/pxURvYHCK2A&hl=en&fs=1&rel=0>

Figure 3.20: This is the example video for using the Words-to-Bits sub-VI.

Download This LabVIEW sub-VI[17]

3.2.6 OFDM Symbol Decoder (Receiver) (LabVIEW)[18]

Input/Outputs and Help



This sub-VI encompasses all of the receiver sub-VIs. Note that frequency offset and channel taps need not be specified if no correction is desired or needed.

For more information about the operation of the individual modules, see their respective descriptions and comments.

Figure 3.21: This is the LabVIEW help and block description for the OFDM Symbol Decoder sub-VI.

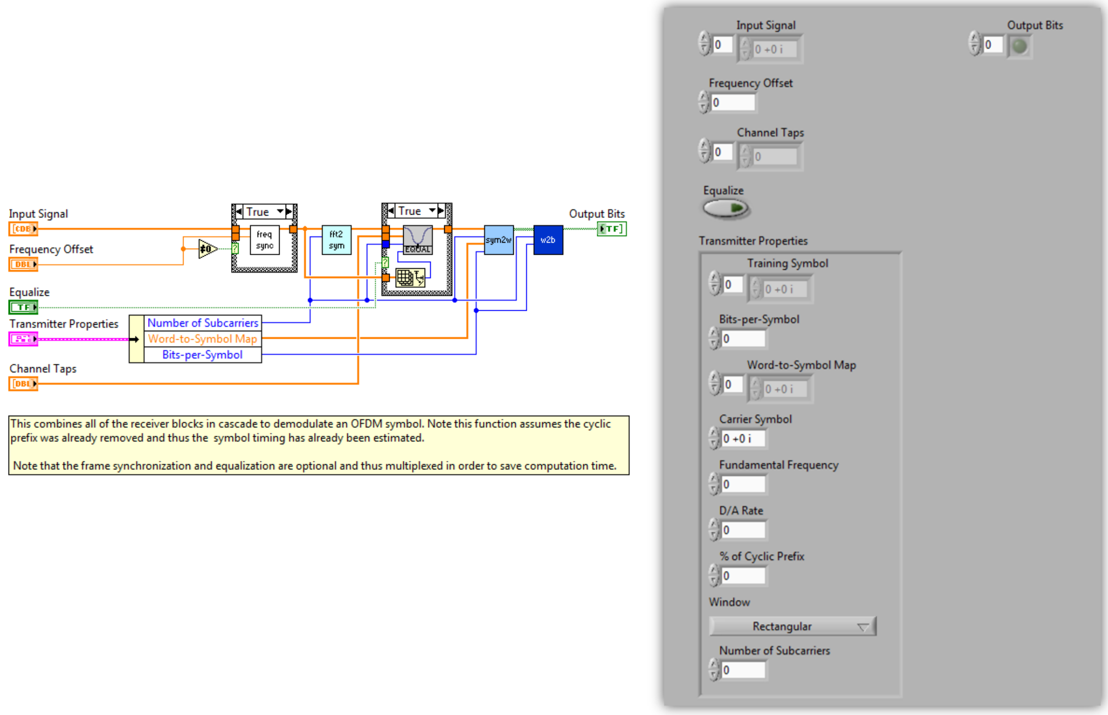
Show above is the LabVIEW sub-VI for receiving generic OFDM symbols to be placed in any real-life communications template. It is designed to be the polar-opposite of the OFDM symbol generator demonstrated

[17] http://cnx.org/content/m34344/latest/w2b.zip
[18] This content is available online at <http://cnx.org/content/m34346/1.1/>.

previously[19] , but can easily be placed in cascade in higher level VIs to tailor to any OFDM system.

Note that this module will be built from previously discussed receiver modules, so if anything is completely alien, check the menu list to the left for information about each sub-VI contained here.

Block Diagram Layout



This combines all of the receiver blocks in cascade to demodulate an OFDM symbol. Note this function assumes the cyclic prefix was already removed and thus the symbol timing has already been estimated.

Note that the frame synchronization and equalization are optional and thus multiplexed in order to save computation time.

Figure 3.22: This is the LabVIEW block diagram for the OFDM Symbol Decoder sub-VI.

As promised, here is our receiver built from all our previous sub-VIs. Note that, as with the OFDM symbol generator, multiple functions are multiplexed in case they're unneeded to save computational complexity.

The VI is built such that it can easily take a transmitter property cluster used in the symbol generation to configure the receiver blocks. Everything seen here has been previously seen prior, and the only pitfall is the use of clusters. We're now ready to test the complete OFDM chain in software to verify the performance of our transceiver. For more information, check the instructional video below and the all software test shown in the left hand column of the collection.

Instructional Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/ZH4dXdYZzSs&hl=en&fs=1&rel=0>

Figure 3.23: This is the instructional video for constructing the OFDM Symbol Decoder sub-VI.

[19]http://cnx.org/content/m34129/latest/

Download This LabVIEW sub-VI[20]
NOTE: This zip file also contains all the sub-VI's shown in this module that this sub-VI uses.

[20]http://cnx.org/content/m34346/latest/ofdmrx.zip

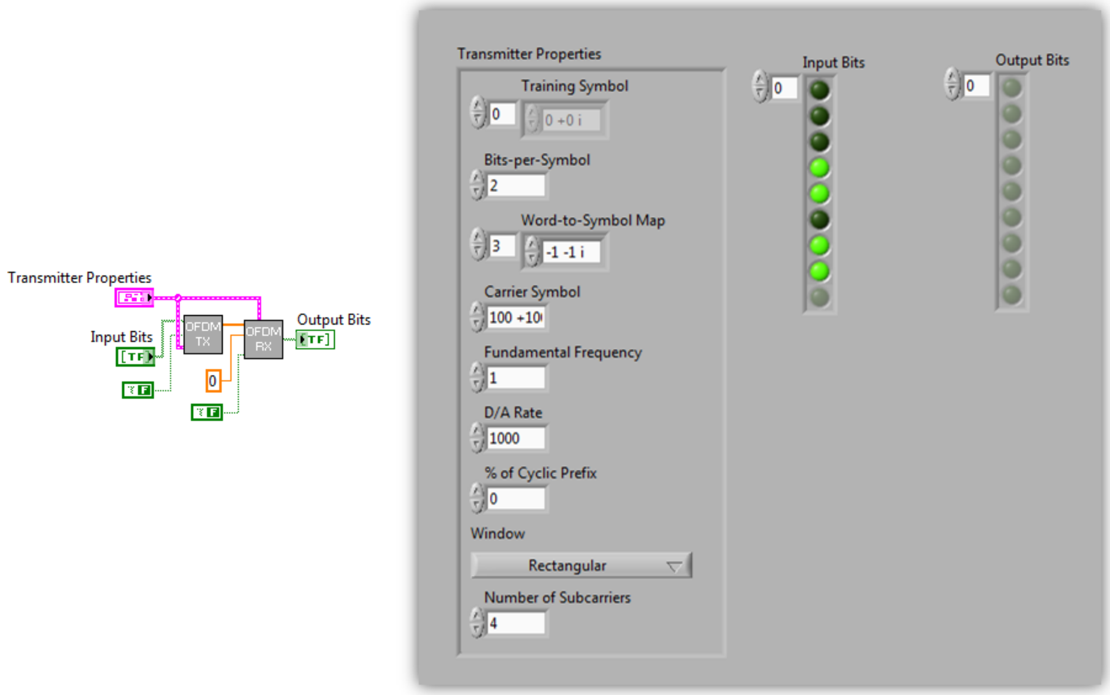
Chapter 4

Tests

4.1 All Software (Tests)[1]

In this module we're ready for a simple test. Before we can place either of our blocks into real-time communications systems, it's prudent to check that they work in an idealized situation. Thus we'd like to test them all in the HOST in LabVIEW

Block Diagram Layout



Transmitter Properties
Training Symbol 0 0 +0 i
Bits-per-Symbol 2
Word-to-Symbol Map 3 -1 -1 i
Carrier Symbol 100 +10
Fundamental Frequency 1
D/A Rate 1000
% of Cyclic Prefix 0
Window Rectangular
Number of Subcarriers 4
Input Bits 0
Output Bits 0

Figure 4.1: This is the LabVIEW block diagram for the All Software Test.

[1]This content is available online at <http://cnx.org/content/m34350/1.1/>.

Above in Figure 1 is the setup for our test. Very straightforward. Check out the video below to see it in action, and download the VI at the bottom.

Example Video

This media object is a Flash object. Please view or download it at
<http://www.youtube.com/v/A4LxvNBZ71Y&hl=en&fs=1&rel=0>

Figure 4.2: This is the example video for using the OFDM Transmitter and Receiver blocks.

Download This LabVIEW sub-VI[2]

[2]http://cnx.org/content/m34350/latest/allsoftware_test.zip

Chapter 5

References

5.1 References[1]

SPECIAL THANKS:

My beautiful wife Aimee, Dr. Christopher Schmitz, Dr. Douglas Jones, Aditya Jain, Steve Jian, Christopher Li.

REFERENCES:

[1] L. Couch, "Digital and Analog Communication Systems," 2007.

[2] B. Farhang-Boroujeny, "Signal Processing Techniques for Software Radios," 2008.

[3] C. Li, C. Schmitz, and A. Muehlfeld, "Building FPGA Communications Projects with LabVIEW," Connexions. August 16, 2009. Available: http://cnx.org/content/m31349/latest. [Accessed: Oct. 16, 2009].

[4] T. Schmidl and D. Cox, "Robust Frequency and Timing Synchronization for OFDM," IEEE Transactions on Communications, vol. 45, no. 12, pp. 1613-1621, Dec. 1997.

[5] P. Koch and R. Prasad, "The Universal Handset," IEEE Spectrum, vol. 36, no. 4, pp. 36-41, Apr. 2009.

[6] C. Langton, "Orthogonal Frequency Division Multiplexing (OFDM) Tutorial," Complex2Real.com: Intuitive Guide to Principles of Communications. 2004. Available: http://www.complextoreal.com/chapters/ofdm2.pdf. [Accessed: Oct. 16, 2009].

[7] H. Minn, M. Zeng, and V. K. Bhargava, "On Timing Offset Estimation for OFDM Systems," IEEE Communications Letters, vol. 4, no. 7, pp. 242-244, Jul. 2000.

[8] C. Schmitz, "ECE 463: Digital Communications Laboratory," ECE Illinois. Spring 2010. Available: http://courses.ece.illinois.edu/ece463/SP10/index.html. [Accessed: May. 1, 2010].

[1]This content is available online at <http://cnx.org/content/m34352/1.1/>.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Fully Configurable OFDM SDR Transceiver in LabVIEW
Our project originally realized an OFDM transmitter on the NI PCI-5640 IF-RIO (intermediate frequency, reconfigurable input/output) transceiver, which contains the Virtex-II Pro Field Programmable Gate Array (FPGA) and the receiver in MATLAB. It was completed in about 8 weeks as a final project for Digital Signal Processing Laboratory (ECE 420) at the University of Illinois at Urbana/Champaign. Using the latest LabView software, we were able to create an incredibly flexible OFDM transceiver that could be scaled up or down with ease. This of course is especially exciting given the prospect of cognitive radio, for which OFDM is a heavy favorite scheme of choice. With the ability to scale the number of subcarriers up and down with little impact on the receiver structure, adapting to open spectrum and conditions becomes much easier. The continuation of this project will culminate in a significant educational Connexions (CNX) module collection complete with written and visual tutorials. YouTUBE videos recorded from Camtasia Studio will close the gap between reading about the project and seeing it come together. In the end, the collection will fully detail how to implement a complete OFDM SDR system with transmitter and receiver. The final product will run on a front-end receiver utilizing National Instruments' PXI chassis populated with the 5660 digital downconverter/high-speed digitizer in conjuction with the 5671 AWG/upconverter. The receiver back-end will be purely software-defined and utilize LabVIEW.