[ECE573]: Advanced Embedded Logic Design
Winter 2025

Project Report 1

Design Development of IEEE 802.11a Architecture on Zedboard

Instructor: Dr. Sumit J Darak
Mentor: Jai Mangal

Ahilan R
MT24170
ahilan24170@iiitd.ac.in

Ippili Saravana Kumar
MT24181
ippili24181@iiitd.ac.in

Ashin VA
MT24177
ashin24177@iiitd.ac.in

Mintu Kumar
2022296
mintu22296@iiitd.ac.in

Shreemann JH
MT24206
shreemann24206@iiitd.ac.in

March 16, 2025

Contents

1 Introduction

Orthogonal Frequency Division Multiplexing (OFDM) is a widely used modulation technique in wireless communication standards such as IEEE 802.11a. It provides high spectral efficiency and robustness against multipath fading and inter-symbol interference (ISI). The implementation consists of two primary stages: transmission (modulation) and reception (demodulation).

The transmitter performs modulation, serial-to-parallel conversion (S/P), IFFT, cyclic prefix insertion, and preamble addition before the signal is sent through the channel. The receiver reconstructs the original data by performing packet detection, frequency offset correction, cyclic prefix removal, FFT, channel estimation, and demodulation.

The figure 1 illustrates the key processing blocks in both transmitter and receiver implementations. The receiver first detects the incoming packet, corrects frequency offsets, estimates the channel response, and finally extracts the transmitted data. These steps ensure reliable wireless communication in the presence of noise and interference.

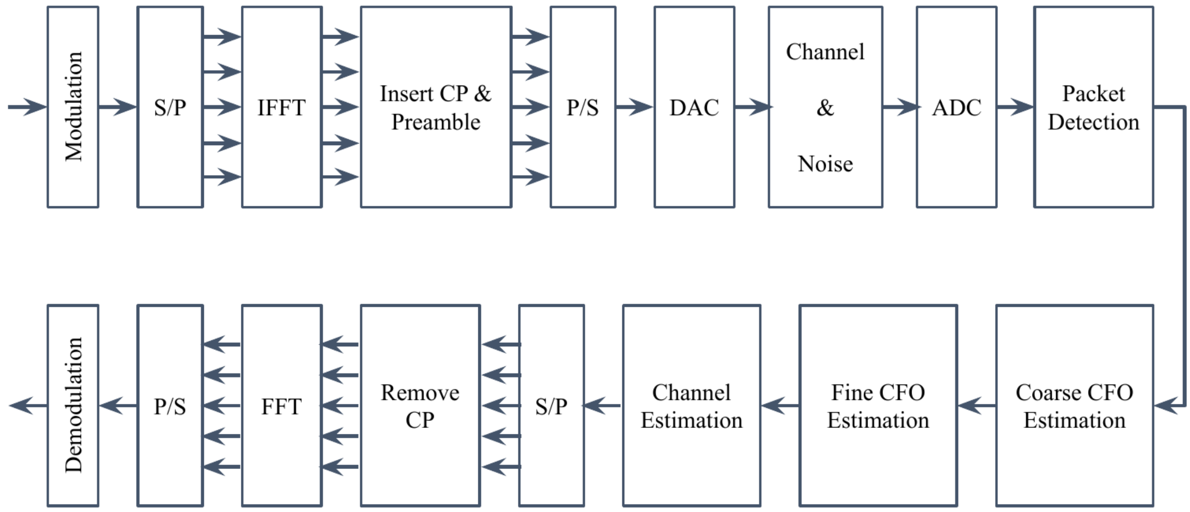

Figure 1: OFDM Block Diagram

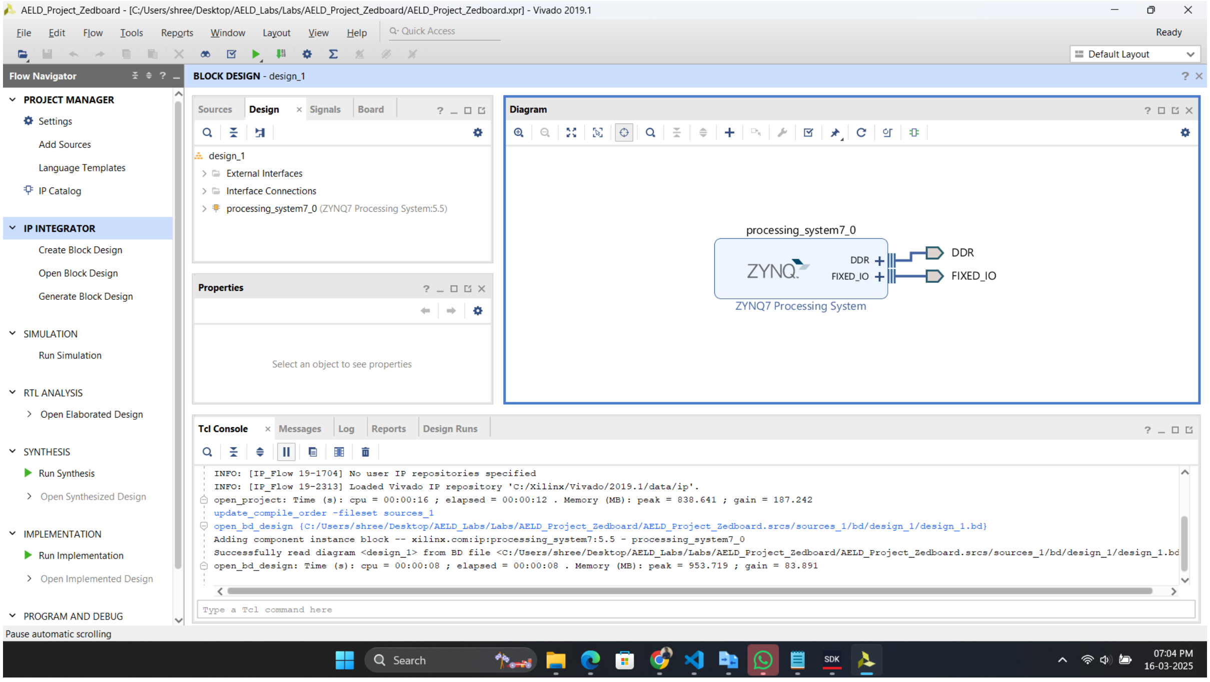

Figure 2: PS Block Design

1.1 Signal Parameters

- FFT Size (N_{FFT}): 64
- Carrier Frequency (f_c): 5 GHz
- Bandwidth: 20 MHz
- Time Period (t_s): 50 ns

2 Transmitter Design

2.1 Short Preamble

- The short preamble is used for packet detection and coarse frequency offset estimation.
- The short preamble sequence S_k is defined in the frequency domain and then transformed into the time domain using an Inverse Fast Fourier Transform (IFFT).
- The short preamble is repeated 10 times to form a 160-sample sequence.

```c
// Function to create the short preamble
void create_short_preamble(complex double *Short_preamble) {
    // Virtual subcarriers (padding zeros)
    complex double virtual_subcarrier[11] = {0};

    // Constructing the frequency-domain preamble (64 elements)
    complex double Short_preamble_slot_Frequency[N_FFT];

    for (int i = 0; i < 6; i++)
        Short_preamble_slot_Frequency[i] = virtual_subcarrier[i];

    for (int i = 0; i < 53; i++)
        Short_preamble_slot_Frequency[i + 6] = S_k[i] * sqrt(13.0 / 6);

    for (int i = 0; i < 5; i++)
        Short_preamble_slot_Frequency[i + 59] = virtual_subcarrier[i];

    // Perform IFFT shift
    fftshift(Short_preamble_slot_Frequency, N_FFT);

    // Time-domain representation of the short preamble
    complex double Short_preamble_slot_Time[N_FFT];
    ifft(Short_preamble_slot_Frequency, Short_preamble_slot_Time, N_FFT);

    // Generate the final Short Preamble sequence (repeat first 16 values 10 times)
    for (int i = 0; i < 10; i++)
        for (int j = 0; j < 16; j++)
            Short_preamble[i * 16 + j] = Short_preamble_slot_Time[j];
}
```

The definition for `fftshift`, `ifft` can be found in appendix.

2.2 Long Preamble

- The long preamble is used for channel estimation and fine frequency offset estimation.
- The long preamble sequence L_k is defined in the frequency domain and transformed into the time domain using IFFT.
- The long preamble is repeated to form a 160-sample sequence.

```c
// Function to create the long preamble
void create_long_preamble(complex double *Long_preamble) {
    // Virtual subcarriers (padding zeros)
    complex double virtual_subcarrier[11] = {0};

    // Constructing the frequency-domain preamble (64 elements)
    complex double Long_preamble_slot_Frequency[N_FFT];

    for (int i = 0; i < 6; i++)
        Long_preamble_slot_Frequency[i] = virtual_subcarrier[i];

    for (int i = 0; i < 53; i++)
        Long_preamble_slot_Frequency[i + 6] = L_k[i];

    for (int i = 0; i < 5; i++)
        Long_preamble_slot_Frequency[i + 59] = virtual_subcarrier[i];

    // Perform IFFT shift
    fftshift(Long_preamble_slot_Frequency, N_FFT);

    // Time-domain representation of the long preamble
    complex double Long_preamble_slot_Time[N_FFT];
    ifft(Long_preamble_slot_Frequency, Long_preamble_slot_Time, N_FFT);

    // Generate the final Long Preamble sequence
    for (int i = 0; i < 32; i++)
        Long_preamble[i] = Long_preamble_slot_Time[i + 32];

    for (int i = 0; i < 64; i++)
        Long_preamble[i + 32] = Long_preamble_slot_Time[i];

    for (int i = 0; i < 64; i++)
        Long_preamble[i + 96] = Long_preamble_slot_Time[i];
}
```

2.3 Payload

- The payload data is a text, which is converted to binary and modulated using QPSK.
- The payload is divided into two frames, each containing 48 QPSK symbols.
- Pilot symbols are inserted into the payload for channel estimation and tracking.
- The payload is transformed into the time domain using IFFT, and a cyclic prefix (CP) is added to mitigate inter-symbol interference (ISI).

```c
// Custom Payload
const char text_1[12] = "HELLO_EMBED!"; //Const 12 char message for 96 bits
const char text_2[12] = "EMBEDDED_SYS"; //Const 12 char message for 96 bits

// Function to create the payload
```

```c
void create_payload(int *data_bits_1, int *data_bits_2, complex double
↪  *data_payload_1, complex double *data_payload_2, complex double
↪  *data_1_TX_payload, complex double *data_2_TX_payload) {
    int data_bits_1[N_BITS], data_bits_2[N_BITS];

    //complex double data_payload_1[48], data_payload_2[48];
    complex double virtual_subcarrier[11] = {0};
    complex double pilot[4] = {1, 1, 1, -1};

   //Converting the text to binary data
    text_to_binary(text_1, data_bits_1);
    text_to_binary(text_2,data_bits_2);

    // Perform QPSK modulation
    qpsk_modulation(data_bits_1, data_payload_1, N_BITS);
    qpsk_modulation(data_bits_2, data_payload_2, N_BITS);

    // Construct frames
    complex double data_frame_1[N_FFT], data_frame_2[N_FFT];
    construct_frame(data_payload_1, data_frame_1, virtual_subcarrier, pilot);
    construct_frame(data_payload_2, data_frame_2, virtual_subcarrier, pilot);

    // Apply FFT shift
    fftshift(data_frame_1, N_FFT);
    fftshift(data_frame_2, N_FFT);

    // Apply IFFT
    complex double data_frame_1_ifft[N_FFT], data_frame_2_ifft[N_FFT];
    ifft(data_frame_1, data_frame_1_ifft, N_FFT);
    ifft(data_frame_2, data_frame_2_ifft, N_FFT);

    // Add cyclic prefix
    for (int i = 0; i < 16; i++) {
        data_1_TX_payload[i] = data_frame_1_ifft[i + 48];
        data_2_TX_payload[i] = data_frame_2_ifft[i + 48];
    }
    for (int i = 0; i < 64; i++) {
        data_1_TX_payload[i + 16] = data_frame_1_ifft[i];
        data_2_TX_payload[i + 16] = data_frame_2_ifft[i];
    }
}
```

The definitions for `text_to_binary`, `qpsk_modulation` and `construct_frame` can be found in appendix.

2.4 Frame Combination

- The transmitted frame is constructed by concatenating the short preamble, long preamble, and the two payload frames.
- The frame is then oversampled by a factor of 2 to prepare it for transmission.

```c
// Function to construct the frame
void construct_frame(complex double *payload, complex double *frame, complex double
↪ *virtual_subcarrier, complex double *pilot) {
    for (int i = 0; i < 6; i++) frame[i] = virtual_subcarrier[i];
    for (int i = 0; i < 5; i++) frame[i + 6] = payload[i];
    frame[11] = pilot[0];
    for (int i = 0; i < 13; i++) frame[i + 12] = payload[i + 5];
    frame[25] = pilot[1];
    for (int i = 0; i < 6; i++) frame[i + 26] = payload[i + 18];
    frame[32] = 0;
    for (int i = 0; i < 6; i++) frame[i + 33] = payload[i + 24];
    frame[39] = pilot[2];
    for (int i = 0; i < 13; i++) frame[i + 40] = payload[i + 30];
    frame[53] = pilot[3];
    for (int i = 0; i < 5; i++) frame[i + 54] = payload[i + 43];
    for (int i = 0; i < 5; i++) frame[i + 59] = virtual_subcarrier[i + 6];
}

// Function to oversample Frame_Tx
void oversampleFrameTx(complex double *Frame_Tx_Oversampled, complex double
↪ *Frame_Tx) {
    for (int i = 0; i < FRAME_TX_OVERSAMP_LENGTH; i++) {
        if (i % OVERSAMPLINGFACTOR == 0) {
            Frame_Tx_Oversampled[i] = Frame_Tx[i / OVERSAMPLINGFACTOR]; // Copy
            ↪ original element
        } else {
            Frame_Tx_Oversampled[i] = 0 + 0 * I; // Insert zero (complex zero)
        }
    }
}
```

2.5 Root Raised Cosine Filter (RRC)

- The transmitted signal is passed through a Root Raised Cosine (RRC) filter to shape the signal and reduce inter-symbol interference.
- The RRC filter is designed with a roll-off factor of 0.5 and a length of 10.

```c
void convolveWithRRC(complex double *Tx_signal, complex double
↪ *Frame_Tx_Oversampled, const double *rrc_filter_tx, int filter_length, int
↪ output_signal_len){
     //Creating the padding signal
   //padding length for each side,ie the number of zeros to be added on each end
   ↪ side.
   int pad_length = filter_length - 1;
   int len_padded = FRAME_TX_OVERSAMP_LENGTH + 2 * pad_length; // Total length of
   ↪ padded signal 1000

   //signal for padding
   complex double inp_sig_padded[len_padded];
   // Fill the padded array with zeros then Frame_Tx_Oversamp values and again
   ↪ zeros
   for (int i = 0; i < filter_length - 1; i++) {
```

```
    inp_sig_padded[i] = 0;
    }
    for (int i = 0; i < FRAME_TX_OVERSAMP_LENGTH; i++) {
    inp_sig_padded[filter_length - 1 + i] = Frame_Tx_Oversampled[i];
    }
    for (int i = 0; i < filter_length - 1; i++) {
    inp_sig_padded[filter_length - 1 + FRAME_TX_OVERSAMP_LENGTH + i] = 0;
    }

    //Initializing empty Tx_signal
     for (int i = 0; i < output_signal_len; i++) {
   Tx_signal[i] = 0;
     }
    //perform convolution between rcc coefff and padded signal
    for (int n = 0; n < output_signal_len; n++) {
     for (int k = 0; k < filter_length; k++) {
          Tx_signal[n] += rrc_filter_tx[k] * inp_sig_padded[n + k]; // Multiply and
          ↪   accumulate
           }
    }
}
```

2.6 Transmission

- The signal is repeated 10 times to simulate continuous transmission.
- The signal is then transmitted over a noisy channel with varying Signal-to-Noise Ratios (SNRs).

```
// Repeating the convoluted Tx_signal(980 samples) 10 times = 9800 samples
int Tx_signal_9800len= output_signal_len*10;
complex double Tx_signal_9800[Tx_signal_9800len];
repeat10times(output_signal_len, Tx_signal, Tx_signal_9800);

// Over The Air Transmission (Channel)
complex double Tx_ota_signal[9800];
double snr_dB_values[NUM_SNR_VALUES] = {5,6,7,8,9,10,11,12,13,14};
// Calculate the average power of the transmitted signal
double Tx_signal_power = 0.0;
for (int i = 0; i < Tx_signal_9800len; i++) {
   Tx_signal_power += creal(Tx_signal_9800[i] * conj(Tx_signal_9800[i])); //
   ↪   |Tx_signal|^2
}
Tx_signal_power /= output_signal_len;

Results results[NUM_SNR_VALUES];
for (int idx_snr_j = 0; idx_snr_j < NUM_SNR_VALUES; idx_snr_j++) {
   double snr_dB = snr_dB_values[idx_snr_j];
   double snr_linear = pow(10, snr_dB / 10.0); //Converting dB to linear scale
   double noise_power = Tx_signal_power / snr_linear;
   double noise_stddev = sqrt(noise_power/2);
   for (int i = 0; i < Tx_signal_9800len; i++) {
       complex double noise = noise_stddev * (((double)rand()/RAND_MAX) +
       ↪   ((double)rand()/RAND_MAX) * I); // generating noise
```

```
        Tx_ota_signal[i] = Tx_signal_9800[i] + noise;  //adding noise
    }
```

3 Receiver Design

3.1 Packet Detection & Selection

- The receiver captures a random segment of the transmitted signal.
- The received signal is passed through an RRC filter to remove out-of-band noise.
- Packet detection is performed by correlating the received signal with a delayed version of itself. The correlation peak indicates the start of a packet.

```
//***********************RECEIVER STARTS************************/
srand(2); // Equivalent to MATLAB's rng('default')
int rx_start = rand() % (Tx_signal_9800len - NP_PACKETS_CAPTURE + 1); //generating a random variable between(0,9800-3000)

//Rx_signal variable which will hold the captured received signal
complex double Rx_signal[NP_PACKETS_CAPTURE];
//printf("\n Randomly starting to capture packets from Channel ...\n");
//creating rx signal of 3000 samples from Tx_ota_signal
for (int i = 0; i < NP_PACKETS_CAPTURE; i++) {
    Rx_signal[i] = Tx_ota_signal[rx_start + i];
}

int Rx_signal_len = sizeof(Rx_signal) / sizeof(Rx_signal[0]);  //calculating length of rx_signal 3000

//Convolve the Rx frame with the RRC filter:-
int rrc_filter_rx_length = sizeof(rrc_filter_rx) / sizeof(rrc_filter_rx[0]); //filterlength  21
int Rx_filtered_signal_len= (Rx_signal_len + rrc_filter_rx_length - 1); //length of output signal ,3020

//Creating empty Rx_filtered_signal, for output
complex double Rx_filtered_signal[Rx_filtered_signal_len];
for (int i = 0; i < Rx_filtered_signal_len; i++) {
    Rx_filtered_signal[i] = 0 + 0 * I;
}

//Creating the padding signal
//padding length for each side,ie the number of zeros to be added on each end side.
int pad_length_rx = rrc_filter_rx_length - 1;
int len_padded_rx = Rx_signal_len + 2 * pad_length_rx; // Total length of padded signal

//Signal for padding rx
complex double inp_sig_padded_rx[len_padded_rx];

// Fill the padded array with zeros then Rx_signal values and again zeros
for (int i = 0; i < rrc_filter_rx_length - 1; i++) {
inp_sig_padded_rx[i] = 0;
}
for (int i = 0; i < Rx_signal_len; i++) {
inp_sig_padded_rx[rrc_filter_tx_length - 1 + i] = Rx_signal[i];
}
```

```
for (int i = 0; i < rrc_filter_tx_length - 1; i++) {
inp_sig_padded_rx[rrc_filter_tx_length - 1 + FRAME_TX_OVERSAMP_LENGTH + i] = 0;
}

//Perform convolution between rcc coefff rx and padded signal rx
for (int n = 0; n < Rx_filtered_signal_len; n++) {
     for (int k = 0; k < rrc_filter_rx_length; k++) {
         Rx_filtered_signal[n] += rrc_filter_rx[k] * inp_sig_padded_rx[n + k]; //
         ↪  Multiply and accumulate
          }
}
 //end of convolution for receiver part.
 //Now we have Rx_filtered_signal which is the result of convolution of coefficient
  ↪  and signal with padded zeroes.
 //We have the Rx_filtered_signal with us

// Prepare output array for normalized correlation.
//Packet Detection
double corr_out[OUT_LENGTH];     //   corr_out: output array of normalized
↪  correlation values (length = out_length)
// Compute normalized correlation using the function.
compute_normalized_correlation(Rx_signal, LEN_RX, DELAY_PARAM, WINDOW_LENGTH,
↪  corr_out, OUT_LENGTH);

// Packet selection.
int packet_idx = packetSelection(corr_out, OUT_LENGTH);
```

The definition for `compute_normalized_correlation` is in appendix.

3.2 Coarse CFO Estimation

- Coarse Carrier Frequency Offset (CFO) estimation is performed using the short preamble.
- Define the length of the short preamble slot: $L = 16$
- Calculate the complex conjugate product:

$$P = \sum_{n=1}^{L} r[n + 5L] \cdot r^*[n + 6L]$$

  where:

  - $r[n]$ is the received frame
  - * denotes the complex conjugate

- Estimate the coarse frequency offset:

$$\hat{f}_{coarse} = -\frac{1}{2\pi L T_s} \cdot atan2\Big(\Im(P), \Re(P)\Big)$$

  where:

  - T_s is the sampling period

- $\Im(P)$ and $\Re(P)$ represent the imaginary and real parts of P, respectively

- Apply the coarse frequency offset correction:

$$r_{coarse}[n] = r[n] \cdot e^{-j2\pi \hat{f}_{coarse} T_s n}$$

for $n = 0, 1, \ldots, 479$.

```c
// Function for Coarse CFO Estimation and Correction
void coarseCFOEstimation(complex double *rx_frame, complex double
↪ *rx_frame_after_coarse, int rx_frame_len) {
    // Calculate the complex conjugate product
    complex double prod_consq_frame_coarse = 0.0;
    for (int i = 0; i < SHORT_PREAMBLE_SLOT_LENGTH; i++) {
        int idx1 = SHORT_PREAMBLE_SLOT_LENGTH * 5 + i;
        int idx2 = SHORT_PREAMBLE_SLOT_LENGTH * 6 + i;
        prod_consq_frame_coarse += rx_frame[idx1] * conj(rx_frame[idx2]);
    }

    // Estimate the coarse frequency offset
    double phase = atan2(cimag(prod_consq_frame_coarse),
    ↪ creal(prod_consq_frame_coarse));
    double freq_coarse_est = (-1.0 / (2.0 * PI * SHORT_PREAMBLE_SLOT_LENGTH *
    ↪ TS_SEC)) * phase;

    // Apply the coarse frequency offset to the received frame
    for (int n = 0; n < rx_frame_len; n++) {
        complex double correction = cexp(-I * 2.0 * PI * freq_coarse_est * TS_SEC *
        ↪ n);
        rx_frame_after_coarse[n] = rx_frame[n] * correction;
    }
}
```

3.3 Fine CFO Estimation

- Fine CFO estimation is performed using the long preamble.
- The CFO is estimated by comparing the phase difference between two consecutive long preamble sequences.
- Define the length of the short preamble slot: $L = 16$
- Calculate the complex conjugate product for fine CFO estimation:

$$P_{fine} = \sum_{n=1}^{64} r_{coarse}[n + 12L] \cdot r^*_{coarse}[n + 16L]$$

where:

- $r_{coarse}[n]$ is the frame after coarse CFO correction
- * denotes the complex conjugate

- Estimate the fine frequency offset:

$$\hat{f}_{fine} = -\frac{1}{2\pi \cdot 64 \cdot T_s} \cdot atan2\Big(\Im(P_{fine}), \Re(P_{fine})\Big)$$

  where:

  - T_s is the sampling period
  - $\Im(P_{fine})$ and $\Re(P_{fine})$ represent the imaginary and real parts of P_{fine}, respectively

- Apply the fine frequency offset correction:

$$r_{fine}[n] = r_{coarse}[n] \cdot e^{-j2\pi \hat{f}_{fine} T_s n}$$

  for $n = 0, 1, \ldots, 479$.u

```c
// Function for Fine CFO Estimation and Correction
void fineCFOEstimation(complex double *rx_frame_after_coarse, complex double
↪ *rx_frame_after_fine, int rx_frame_len) {
    // Calculate the complex conjugate product for fine CFO estimation
    complex double prod_consq_frame_fine = 0.0;
    for (int i = 0; i < SHORT_PREAMBLE_SLOT_LENGTH * 4; i++) {
        int idx1 = SHORT_PREAMBLE_SLOT_LENGTH * 12 + i;
        int idx2 = SHORT_PREAMBLE_SLOT_LENGTH * 16 + i;
        prod_consq_frame_fine += rx_frame_after_coarse[idx1] *
        ↪ conj(rx_frame_after_coarse[idx2]);
    }

    // Estimate the fine frequency offset
    double phase = atan2(cimag(prod_consq_frame_fine),
    ↪ creal(prod_consq_frame_fine));
    double freq_fine_est = (-1.0 / (2.0 * PI * 64 * TS_SEC)) * phase;

    // Apply the fine frequency offset to the received frame
    for (int n = 0; n < rx_frame_len; n++) {
        complex double correction = cexp(-I * 2.0 * PI * freq_fine_est * TS_SEC *
        ↪ n);
        rx_frame_after_fine[n] = rx_frame_after_coarse[n] * correction;
    }
}
```

3.4 Channel Estimation

- Channel estimation is performed using the long preamble.
- The channel frequency response is estimated by averaging the FFT of two long preamble sequences and multiplying by the conjugate of the known long preamble sequence.
- The channel estimate is then transformed back to the time domain using IFFT.
- `fft` definition can be found in appendix

```c
void channelEstimation(complex double *rx_frame_after_fine, complex double
↪  *H_est_used_for_fft, complex double *H_est, complex double *H_est_time){
    double complex Long_preamble_1[N_FFT]; // Output for Long_preamble_1
    double complex Long_preamble_2[N_FFT]; // Output for Long_preamble_2

    for (int i = 0; i < N_FFT; i++) {
      Long_preamble_1[i] = rx_frame_after_fine[SHORT_PREAMBLE_SLOT_LENGTH*12 + i];
      Long_preamble_2[i] = rx_frame_after_fine[SHORT_PREAMBLE_SLOT_LENGTH*16 + i];
    }

    double complex Long_preamble_1_After_FFT[N_FFT];
    double complex Long_preamble_2_After_FFT[N_FFT];

    // Perform FFT on Long_preamble_1
    fft(Long_preamble_1, Long_preamble_1_After_FFT, N_FFT);
    fftshift(Long_preamble_1_After_FFT, N_FFT);
    // Perform FFT on Long_preamble_2
    fft(Long_preamble_2, Long_preamble_2_After_FFT, N_FFT);
    fftshift(Long_preamble_2_After_FFT, N_FFT);

   //VALUES OF SLOT FREQUENCY
   complex double Long_preamble_slot_Frequency[N_FFT];
   // Virtual subcarriers (padding zeros)
   complex double virtual_subcarrier[11] = {0};

   for (int i = 0; i < 6; i++)
   Long_preamble_slot_Frequency[i] = virtual_subcarrier[i];
   for (int i = 0; i < 53; i++)
   Long_preamble_slot_Frequency[i + 6] = L_k[i];
   for (int i = 0; i < 5; i++)
   Long_preamble_slot_Frequency[i + 59] = virtual_subcarrier[i];

   // Perform the computation
   for (int i = 0; i < 64; i++) {
   H_est[i] = 0.5 * (Long_preamble_1_After_FFT[i] + Long_preamble_2_After_FFT[i]) *
   ↪  conj(Long_preamble_slot_Frequency[i]);
   }
   for(int i=0;i<N_FFT;i++){
   H_est_used_for_fft[i] = H_est[i];
   }

   fftshift(H_est_used_for_fft, N_FFT);
   ifft(H_est_used_for_fft, H_est_time, N_FFT);
}
```

3.5 One-Tap Equalizer

- The received payload is equalized using the estimated channel response.
- The equalizer compensates for the channel effects by dividing the received signal in the frequency domain by the channel estimate.

```c
void oneTapEqualizer(complex double *rx_frame_after_fine, complex double *H_est,
↪  complex double *RX_Payload_1_Frequency, complex double *RX_Payload_2_Frequency,
↪  complex double *RX_Payload_1_Frequency_Equalizer, complex double
↪  *RX_Payload_2_Frequency_Equalizer){
        // Step 1: Extract RX_Payload_1_time (elements 321 to 400)
        complex double RX_Payload_1_time[80];
        for (int i = 0; i < 80; i++) {
            RX_Payload_1_time[i] = rx_frame_after_fine[320 + i];
        }

        // Step 2: Remove cyclic prefix (first 16 elements)
        complex double RX_Payload_1_no_CP[N_FFT];
        for (int i = 0; i < N_FFT; i++) {
            RX_Payload_1_no_CP[i] = RX_Payload_1_time[16 + i];
        }

        // Step 3: Perform FFT and FFT shift
        fft(RX_Payload_1_no_CP, RX_Payload_1_Frequency, N_FFT);
        fftshift(RX_Payload_1_Frequency, N_FFT);

        // Step 4: Equalize the frequency-domain data
        for (int i = 0; i < N_FFT; i++) {
            RX_Payload_1_Frequency_Equalizer[i] = RX_Payload_1_Frequency[i] /
            ↪  H_est[i];
        }

        // Step 1: Extract RX_Payload_2_time (elements 401 to 480)
        complex double RX_Payload_2_time[80];
        for (int i = 0; i < 80; i++) {
            RX_Payload_2_time[i] = rx_frame_after_fine[400 + i];
        }

        // Step 2: Remove cyclic prefix (first 16 elements)
        complex double RX_Payload_2_no_CP[N_FFT];
        for (int i = 0; i < N_FFT; i++) {
            RX_Payload_2_no_CP[i] = RX_Payload_2_time[16 + i];
        }

        // Step 3: Perform FFT and FFT shift
        fft(RX_Payload_2_no_CP, RX_Payload_2_Frequency, N_FFT);
        fftshift(RX_Payload_2_Frequency, N_FFT);

        // Step 4: Equalize the frequency-domain data
        for (int i = 0; i < N_FFT; i++) {
            RX_Payload_2_Frequency_Equalizer[i] = RX_Payload_2_Frequency[i] /
            ↪  H_est[i];
        }
}
```

3.6 De-Mapping and AGC

- The equalized payload is de-mapped by removing the pilot symbols and extracting the data symbols.
- The de-mapped symbols are then passed through an Automatic Gain Control (AGC) block to normalize the signal amplitude.

```
void demapping_RX_Payload(complex double *RX_Payload_Frequency,
    complex double *RX_Payload_Frequency_Equalizer,
    complex double *RX_Payload_no_Equalizer,
    complex double *RX_Payload_no_pilot)
{
// Define the relevant indices to extract
const int indices[] = {6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
↪  23, 24,
     26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 40, 41, 42, 43, 44,
     45, 46, 47, 48, 49, 50, 51, 52, 54, 55, 56, 57, 58};

int num_indices = sizeof(indices) / sizeof(indices[0]);

// Process both outputs in a single loop
for (int i = 0; i < num_indices; i++) {
int idx = indices[i];
RX_Payload_no_Equalizer[i] = RX_Payload_Frequency[idx];
RX_Payload_no_pilot[i] = RX_Payload_Frequency_Equalizer[idx];
}
}

void Rx_Payload_AGC(complex double *RX_Payload_Final, complex double
↪  *RX_Payload_no_pilot) {
    const double scale_factor = 1.0 / M_SQRT2; // Precompute 1/sqrt(2)

    for (int idx = 0; idx < 48; idx++) {
        // Extract real and imaginary parts once
        double real_part = creal(RX_Payload_no_pilot[idx]);
        double imag_part = cimag(RX_Payload_no_pilot[idx]);

        // Apply decision logic using ternary operators (avoids redundant checks)
        double mapped_real = (real_part > 0) ? scale_factor : (real_part < 0) ?
        ↪  -scale_factor : 0;
        double mapped_imag = (imag_part > 0) ? scale_factor : (imag_part < 0) ?
        ↪  -scale_factor : 0;

        // Assign the computed value directly
        RX_Payload_Final[idx] = mapped_real + I * mapped_imag;
    }
}
```

3.7 QPSK Demodulation

- The de-mapped symbols are demodulated to recover the original bits.
- The demodulation process involves mapping the received QPSK symbols back to their corresponding bit pairs.

```
void demodulation_Rx_Payload(int RX_Payload_Final_len, complex double
↪  *RX_Payload_Final, double *RX_Payload_demod,int RX_Payload_demod_len){
    for (int i = 0; i < RX_Payload_demod_len; i++) {
        RX_Payload_demod[i] = 0 + 0 * I; // Initialize to zero (Preallocate the
        ↪  demodulated bits array for better performance)
    }
    // Loop through each symbol
    for (int i = 0; i < RX_Payload_Final_len; i++) {
        // Extract real and imaginary parts of the current symbol
        double RX_Payload_1_demod_real = creal(RX_Payload_Final[i]);
        double RX_Payload_1_demod_imag = cimag(RX_Payload_Final[i]);

        // Determine bits based on QPSK mapping
        if (RX_Payload_1_demod_real > 0 && RX_Payload_1_demod_imag > 0) {
            // Mapping for 00 -> 0.707 + i*0.707
            RX_Payload_demod[2 * i] = 0;
            RX_Payload_demod[2 * i + 1] = 0;
        } else if (RX_Payload_1_demod_real < 0 && RX_Payload_1_demod_imag > 0) {
            // Mapping for 01 -> -0.707 + i*0.707
            RX_Payload_demod[2 * i] = 0;
            RX_Payload_demod[2 * i + 1] = 1;
        } else if (RX_Payload_1_demod_real < 0 && RX_Payload_1_demod_imag < 0) {
            // Mapping for 10 -> -0.707 - i*0.707
            RX_Payload_demod[2 * i] = 1;
            RX_Payload_demod[2 * i + 1] = 0;
        } else if (RX_Payload_1_demod_real > 0 && RX_Payload_1_demod_imag < 0) {
            // Mapping for 11 -> 0.707 - i*0.707
            RX_Payload_demod[2 * i] = 1;
            RX_Payload_demod[2 * i + 1] = 1;
        }
    }
}
```

4 Performance Metrics

4.1 Error Vector Magnitude (EVM)

- EVM is calculated both before and after the AGC block.

$$e[i] = r[i] - s[i]$$

  where:

  - $e[i]$ = error vector (difference between received and transmitted symbols)
  - $r[i]$ = received symbol at index i (without pilot)
  - $s[i]$ = transmitted (modulated) symbol at index i

- EVM measures the difference between the transmitted and received symbols, normalized by the power of the transmitted symbols.

$$EVM = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}|e[i]|^2}}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}|s[i]|^2}}$$

  where N is the total number of symbols.

- EVM is expressed in dB and provides an indication of the signal quality.

$$EVM_{dB} = 20\log_{10}(EVM)$$

4.2 Bit Error Rate (BER)

- Error bits are calculated by comparing the received bits with the original transmitted bits.

$$Error\_bits = \sum_{i=1}^{N_1}|sgn(d_1[i] - r_1[i])| + \sum_{i=1}^{N_2}|sgn(d_2[i] - r_2[i])|$$

  where:

  - $d_1[i]$ = i-th bit of $data\_payload_1$
  - $r_1[i]$ = i-th bit of $rx\_Payload_1\_demod$
  - $d_2[i]$ = i-th bit of $data\_payload_2$
  - $r_2[i]$ = i-th bit of $rx\_Payload_2\_demod$
  - N_1 = length of $data\_payload_1$
  - N_2 = length of $data\_payload_2$

- BER is the ratio of the total number of bit errors to the total number of transmitted bits:

$$BER = \frac{Error\_bits}{N_1 + N_2}$$

- BER is plotted against SNR to evaluate the performance of the system under different noise conditions.

4.3 Code

4.3.1 EVM Before AGC

```
// EVM Calculation Before AGC
// Allocate an array for the concatenated error vector (size = 2*N)

double complex error_vector[N_BITS];

// Calculate error_vector = [RX_Payload_1_no_pilot, RX_Payload_2_no_pilot] -
↪  [data_payload_1_mod, data_payload_2_mod]
for (int i = 0; i < N_BITS/2; i++)
{
    error_vector[i] = RX_Payload_1_no_pilot[i] - data_payload_1_mod[i];
    error_vector[i + N_BITS/2] = RX_Payload_2_no_pilot[i] - data_payload_2_mod[i];
}

// Calculate sum of squared magnitudes for the error vector and for the transmitted
↪  symbols
double sum_error = 0.0;
double sum_ref = 0.0;
for (int i = 0; i < N_BITS/2; i++)
{
    sum_error += pow(cabs(error_vector[i]), 2);
    sum_ref += pow(cabs(data_payload_1_mod[i]), 2);
}
for (int i = 0; i < N_BITS/2; i++)
{
    sum_error += pow(cabs(error_vector[i + N_BITS/2]), 2);
    sum_ref += pow(cabs(data_payload_2_mod[i]), 2);
}

// Calculate EVM as the RMS value of error magnitude normalized by RMS value of
↪  transmitted symbols
double evm = sqrt(sum_error / N_BITS) / sqrt(sum_ref / N_BITS);

// Convert EVM to dB
double evm_dB = 20 * log10(evm);
```

4.3.2 EVM After AGC

```
// EVM Calculation After AGC

double complex error_vector_AGC[N_BITS];
// Calculate error_vector_AGC = [RX_Payload_1_Final, RX_Payload_2_Final] -
↪  [data_payload_1_mod, data_payload_2_mod]
for (int i = 0; i < N_BITS/2; i++)
{
    error_vector_AGC[i] = RX_Payload_1_Final[i] - data_payload_1_mod[i];
    error_vector_AGC[i + N_BITS/2] = RX_Payload_2_Final[i] - data_payload_2_mod[i];
}
// Calculate the sum of squared magnitudes for the error vector and the transmitted
↪  symbols
sum_error = 0.0;
```

```
sum_ref = 0.0;

for (int i = 0; i < N_BITS/2; i++)
{
        sum_error += pow(cabs(error_vector_AGC[i]), 2);
        sum_ref += pow(cabs(data_payload_1_mod[i]), 2);
}
for (int i = 0; i < N_BITS/2; i++)
{
        sum_error += pow(cabs(error_vector_AGC[i + N_BITS/2]), 2);
        sum_ref += pow(cabs(data_payload_2_mod[i]), 2);
}

// Calculate EVM as the RMS error normalized by the RMS of the transmitted symbols
double evm_AGC = sqrt(sum_error / N_BITS) / sqrt(sum_ref / N_BITS);
// Convert EVM to dB
double evm_AGC_dB = 20 * log10(evm_AGC);
```

4.3.3 BER

```
// Calculate the number of error bits.
int Error_bits = 0;
for (int i = 0; i < 96; i++)
{
    // Using sign_func to check if there is a difference.
    if (abs(sign_func(data_bits_1[i] - RX_Payload_1_demod[i])) == 1)
    {
        Error_bits++;
    }
}
for (int i = 0; i < 96; i++)
{
    if (abs(sign_func(data_bits_2[i] - RX_Payload_2_demod[i])) == 1)
    {
        Error_bits++;
    }
}

// Calculate the Bit Error Rate (BER)
double BER = (double)Error_bits / (96 + 96);
```

5 Results and Comparison

5.1 EVM Comparison

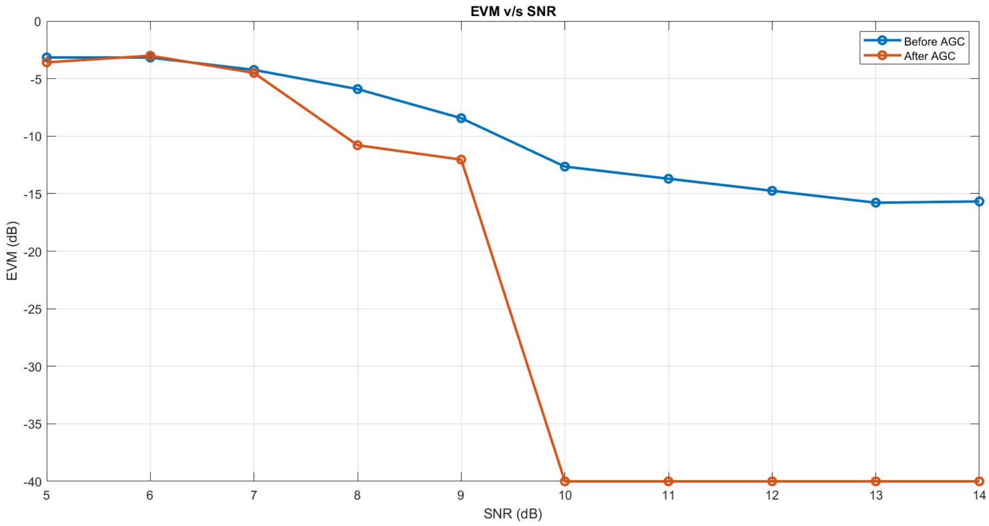

Figure 3: EVM Matlab Ouput

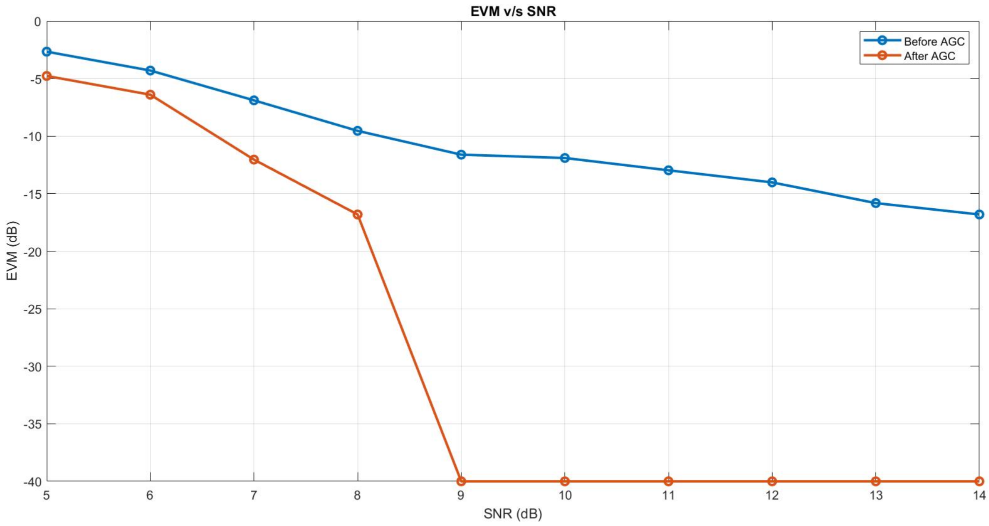

Figure 4: EVM PS Output

5.2 BER Comparison

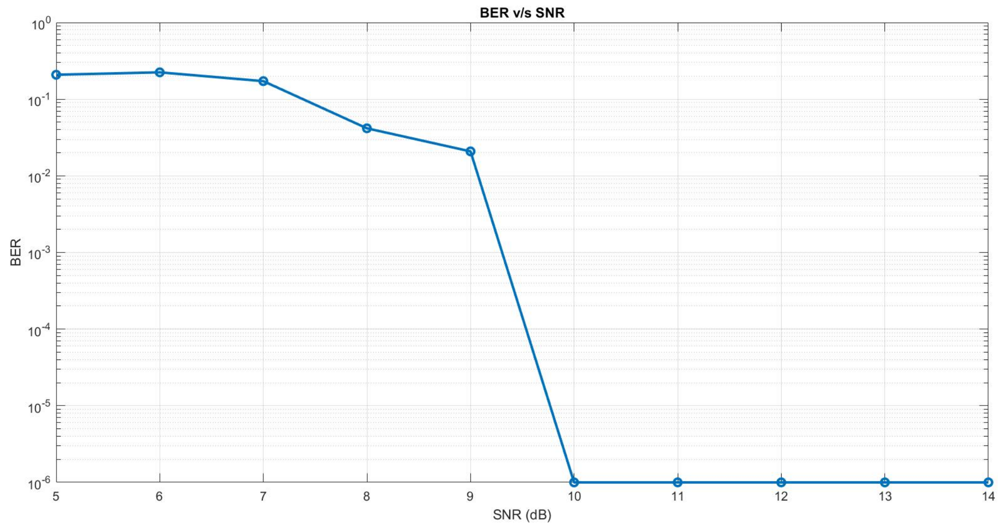

Figure 5: BER Matlab Ouput

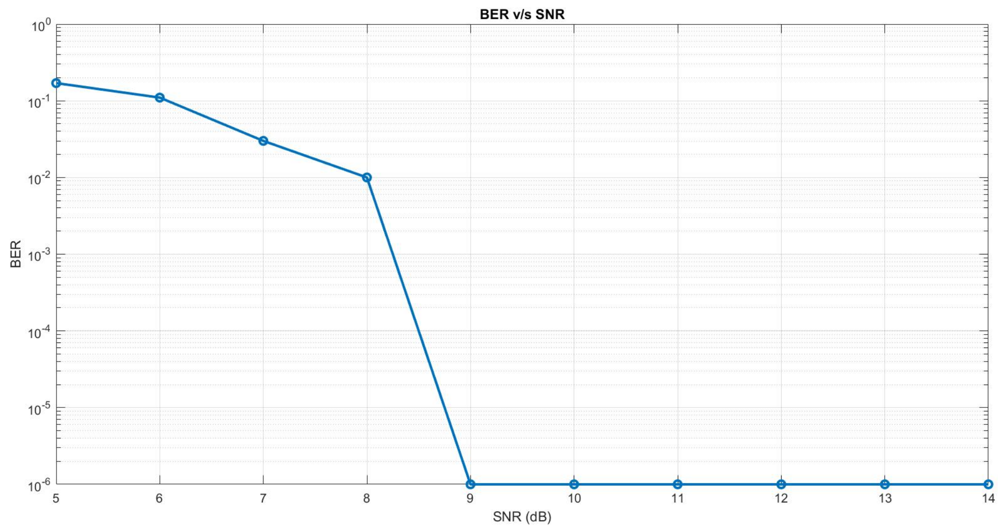

Figure 6: BER PS Output

- As the SNR improves, we can see that the BER and EVM gets better
- We could also see this from the JTAG ouput of the PS

5.3 Timing and Output

```
JTAG-based Hyperterminal.
Connected to JTAG-based Hyperterminal over TCP port : 51517
(using socket : sock620)
Help :
Terminal requirements :
  (i) Processor's STDOUT is redirected to the ARM DCC/MDM UART
  (ii) Processor's STDIN is redirected to the ARM DCC/MDM UART.
       Then, text input from this console will be sent to DCC/MDM's
          UART port.
  NOTE: This is a line-buffered console and you have to press "Enter"
        to send a string of characters to DCC/MDM.

Printing the output for SNR: 5.0 dB
 Packet found!

 Decoded text_1: H OO_EMBFx

 Decoded text_2:   MBEwDuD_SY  ^

 ****************************************************************
Printing the output for SNR: 6.0 dB
 Packet found!

 Decoded text_1: 8ELL_E}BED-

 Decoded text_2:  E    r E    D DSYS

 ****************************************************************
Printing the output for SNR: 7.0 dB
 Packet found!

 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EBEDDED_SYS

 ****************************************************************
Printing the output for SNR: 8.0 dB
 Packet found!

 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EBEDDED_SYS

 ****************************************************************
Printing the output for SNR: 9.0 dB
 Packet found!

 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EMBEDDED_SYS

 ****************************************************************
Printing the output for SNR: 10.0 dB
 Packet found!
```

```
 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EMBEDDED_SYS

 ***************************************************************
Printing the output for SNR: 11.0 dB
 Packet found!

 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EMBEDDED_SYS

 ***************************************************************
Printing the output for SNR: 12.0 dB
 Packet found!

 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EMBEDDED_SYS

 ***************************************************************
Printing the output for SNR: 13.0 dB
 Packet found!

 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EMBEDDED_SYS

 ***************************************************************
Printing the output for SNR: 14.0 dB
 Packet found!

 Decoded text_1: HELLO_EMBED!

 Decoded text_2: EMBEDDED_SYS

 ***************************************************************
EVM_in_dB:     -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
   -12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB:  -4.51 , -5.67 , -16.81 , -16.81 , -40.00 , -40.00 ,
   -40.00 , -40.00 , -40.00 , -40.00
BER:   0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 ,
   0.00
Execution Time for Creating the Frame_TX  : 0.885058 ms
Execution Time for Convolution in Tx  : 1.770092 ms
Execution Time for Transmitter_time : 3.031458 ms
Execution Time for Rx_Packet_Detection  : 6.160399 ms
Execution Time for Rx_Packet_Selection  : 5.240719 ms
Execution Time for RX_Channel_Estimation  : 0.115079 ms
Execution Time for Receiver_Time  : 12.315161 ms
Execution Time for Total_Execution_till_EVM_BER_Calculation  :
   2280.049072 ms
Execution Time for PS  : 2.368793 s
```

5.3.1 Optimized and (Without Print Statements)

```
EVM_in_dB:      -1.80 , -3.81 , -6.80 , -9.78 , -11.53 , -11.93 ,
    -12.97 , -14.01 , -15.72 , -16.72
EVM_AGC_in_dB:  -4.51 , -5.67 , -16.81 , -16.81 , -40.00 , -40.00 ,
    -40.00 , -40.00 , -40.00 , -40.00
BER:    0.17 , 0.13 , 0.01 , 0.01 , 0.00 , 0.00 , 0.00 , 0.00 , 0.00 ,
    0.00
Execution Time for Creating the Frame_TX  : 0.340785 ms
Execution Time for Convolution in Tx  : 1.800985 ms
Execution Time for Transmitter_time : 2.518652 ms
Execution Time for Rx_Packet_Detection  : 0.393187 ms
Execution Time for Rx_Packet_Selection  : 0.012506 ms
Execution Time for RX_Channel_Estimation  : 0.072431 ms
Execution Time for Receiver_Time  : 1.250018 ms
Execution Time for Total_Execution_till_EVM_BER_Calculation  :
    154.128616 ms
Execution Time for PS  : 0.185490 s
```

5.3.2 Screenshots

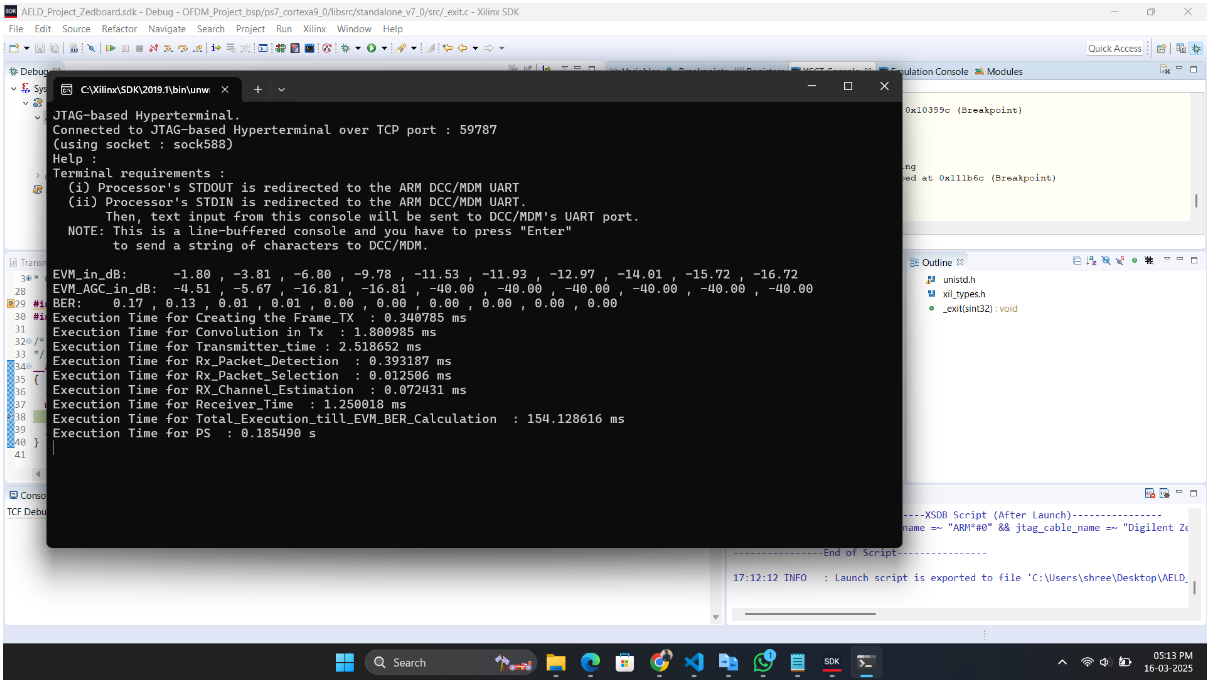

Figure 7: Optimized Output Timing - without print statements

6 Transceiver Implementation Main Code

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include <XTime_l.h>
#include "data.h"
#include "short_preamble.h"
#include "long_preamble.h"
#include "data_payload.h"
#include "correlation.h"
#include "pkt_selection.h"
#include "math.h"
int sign_func(double x)
{
    if (x > 0)
        return 1;
    else if (x < 0)
        return -1;
    else
        return 0;
}

// Structure to store results for each SNR index.
typedef struct
{
    double evm;
    double evm_AGC;
    double ber;
} Results;
void do_main_function(float *frame_tx_time,float *convolution_tx_time,float
↪ *total_tx_time,float *detect_time,float *select_time,float
↪ *estimation_time,float *total_rx_time, float *TOTAL_EXECUTION_TIME){
   complex double Short_preamble[160];
   complex double Long_preamble[160];
   int data_bits_1[N_BITS];
   int data_bits_2[N_BITS];
   complex double data_payload_1_mod[48], data_payload_2_mod[48];
   complex double data_1_TX_payload[80], data_2_TX_payload[80];
   complex double Frame_Tx[FRAME_TX_LEN];
   XTime Total_TX_Start, Total_TX_Stop;
   XTime Frame_Start,Frame_Stop;
      XTime TOTAL_START, TOTAL_STOP;


//****Creating short_preamble, long_preamble, data_payloads and
↪ Frame_Tx*************//
XTime_GetTime(&TOTAL_START);
XTime_GetTime(&Total_TX_Start);
XTime_GetTime(&Frame_Start);
create_short_preamble(Short_preamble);
create_long_preamble(Long_preamble);
create_payload(data_bits_1, data_bits_2, data_payload_1_mod, data_payload_2_mod,
↪ data_1_TX_payload, data_2_TX_payload);
```

```
create_frame_tx(Short_preamble, Long_preamble, data_1_TX_payload, data_2_TX_payload, Frame_Tx);
XTime_GetTime(&Frame_Stop);
*frame_tx_time = (float)1.0 * (Frame_Stop - Frame_Start) / (COUNTS_PER_SECOND/1000000);
//******Oversampling Frame_Tx  (480*2)****************************************//
complex double Frame_Tx_Oversampled[FRAME_TX_OVERSAMP_LENGTH];
oversampleFrameTx(Frame_Tx_Oversampled,Frame_Tx);

//********Convolution with RRC**************************************************//
XTime Convolution_TX_Start,Convolution_TX_Stop;
XTime_GetTime(&Convolution_TX_Start);
int rrc_filter_tx_length = sizeof(rrc_filter_tx) / sizeof(rrc_filter_tx[0]); //filterlength 21
int output_signal_len= (FRAME_TX_OVERSAMP_LENGTH + rrc_filter_tx_length - 1); //length of output signal ,980
complex double Tx_signal[output_signal_len]; //980 Tx_signal samples
convolveWithRRC(Tx_signal, Frame_Tx_Oversampled, rrc_filter_tx, rrc_filter_tx_length, output_signal_len);
XTime_GetTime(&Convolution_TX_Stop);
*convolution_tx_time = (float)1.0 * (Convolution_TX_Stop - Convolution_TX_Start) / (COUNTS_PER_SECOND/1000000);
// Repeating the convoluted Tx_signal(980 samples) 10 times to generate 9800 samples
    int Tx_signal_9800len= output_signal_len*10;
 complex double Tx_signal_9800[Tx_signal_9800len];
 repeat10times(output_signal_len, Tx_signal, Tx_signal_9800);
    XTime_GetTime(&Total_TX_Stop);
    *total_tx_time  =  (float)1.0 * (Total_TX_Stop - Total_TX_Start) / (COUNTS_PER_SECOND/1000000);
//***Over The Air Transmission (Channel)***//
  complex double Tx_ota_signal[9800];
     double snr_dB_values[NUM_SNR_VALUES] = {5,6,7,8,9,10,11,12,13,14};
  // Calculate the average power of the transmitted signal
 double Tx_signal_power = 0.0;
 for (int i = 0; i < Tx_signal_9800len; i++) {
    Tx_signal_power += creal(Tx_signal_9800[i] * conj(Tx_signal_9800[i])); // |Tx_signal|^2
 }
 Tx_signal_power /= output_signal_len;

 Results results[NUM_SNR_VALUES];
 for (int idx_snr_j = 0; idx_snr_j < NUM_SNR_VALUES; idx_snr_j++) {
    double snr_dB = snr_dB_values[idx_snr_j];
    double snr_linear = pow(10, snr_dB / 10.0); //Converting SNR from dB to linear scale
    double noise_power = Tx_signal_power / snr_linear;
    double noise_stddev = sqrt(noise_power/2);
    for (int i = 0; i < Tx_signal_9800len; i++) {
        complex double noise = noise_stddev * (((double)rand()/RAND_MAX) + ((double)rand()/RAND_MAX) * I); // generating noise
        Tx_ota_signal[i] = Tx_signal_9800[i] + noise;  //adding noise
    }

  //  printf("Printing the output for SNR: %.1lf dB \n ", snr_dB_values[idx_snr_j]);
```

```
    //************************RECEIVER STARTS*************************/
   XTime Total_Rx_Start, Total_Rx_Stop;
   XTime_GetTime(&Total_Rx_Start);
    srand(2); // Equivalent to MATLAB's rng('default')
    int rx_start = rand() % (Tx_signal_9800len - NP_PACKETS_CAPTURE + 1);
    ↪ //generating a random variable between(0,9800-3000)
       //printf("rx_start_value_randomly generated: %d \n",rx_start);

    //Rx_signal variable which will hold the captured received signal
       complex double Rx_signal[NP_PACKETS_CAPTURE];
    //printf("\n Randomly starting to capture packets from Channel ...\n");
    //creating rx signal of 3000 samples from Tx_ota_signal
    for (int i = 0; i < NP_PACKETS_CAPTURE; i++) {
        Rx_signal[i] = Tx_ota_signal[rx_start + i];
    }

    int Rx_signal_len = sizeof(Rx_signal) / sizeof(Rx_signal[0]);  //calculating
    ↪  length of rx_signal 3000

    //printf("Rx_Signal_length: %d \n", Rx_signal_len);


     //Convolve the Rx frame with the RRC filter:-
     int rrc_filter_rx_length = sizeof(rrc_filter_rx) / sizeof(rrc_filter_rx[0]);
     ↪ //filterlength  21
     //printf("rrc_filter_rx_length: %d \n", rrc_filter_rx_length);

     int Rx_filtered_signal_len= (Rx_signal_len + rrc_filter_rx_length - 1);
     ↪ //length of output signal ,3020
     //printf("Rx_filtered_signal_len: %d \n", Rx_filtered_signal_len);

      //Creating empty Rx_filtered_signal, for output
      complex double Rx_filtered_signal[Rx_filtered_signal_len];
      for (int i = 0; i < Rx_filtered_signal_len; i++) {
          Rx_filtered_signal[i] = 0 + 0 * I;
      }


      //Creating the padding signal
      //padding length for each side,ie the number of zeros to be added on each
      ↪  end side.
       int pad_length_rx = rrc_filter_rx_length - 1;


       int len_padded_rx = Rx_signal_len + 2 * pad_length_rx; // Total length of
       ↪  padded signal
     //  printf("len_padded_rx: %d \n", len_padded_rx);


       //Signal for padding rx
       complex double inp_sig_padded_rx[len_padded_rx];


       // Fill the padded array with zeros then Rx_signal values and again zeros
       for (int i = 0; i < rrc_filter_rx_length - 1; i++) {
```

```
        inp_sig_padded_rx[i] = 0;
        }
        for (int i = 0; i < Rx_signal_len; i++) {
        inp_sig_padded_rx[rrc_filter_tx_length - 1 + i] = Rx_signal[i];
        }
        for (int i = 0; i < rrc_filter_tx_length - 1; i++) {
        inp_sig_padded_rx[rrc_filter_tx_length - 1 + FRAME_TX_OVERSAMP_LENGTH + i]
        ↪  = 0;
        }



          //Perform convolution between rcc coefff rx and padded signal rx
         for (int n = 0; n < Rx_filtered_signal_len; n++) {
              for (int k = 0; k < rrc_filter_rx_length; k++) {
                  Rx_filtered_signal[n] += rrc_filter_rx[k] * inp_sig_padded_rx[n +
                  ↪  k]; // Multiply and accumulate
                   }
         }
          //end of convolution for receiver part.
          //Now we have Rx_filtered_signal which is the result of convolution of
          ↪  coefficient and signal with padded zeroes.
          //We have the Rx_filtered_signal with us
//****************************************************************************//
// Prepare output array for normalized correlation.
//Packet Detection
     XTime Detect_Start, Detect_Stop;
     XTime_GetTime(&Detect_Start);
     double corr_out[OUT_LENGTH];

     //   corr_out: output array of normalized correlation values (length =
     ↪  out_length)
     // Compute normalized correlation using the function.
     compute_normalized_correlation(Rx_signal, LEN_RX, DELAY_PARAM, WINDOW_LENGTH,
     ↪  corr_out, OUT_LENGTH);
     XTime_GetTime(&Detect_Stop);
     *detect_time = (float)1.0 * (Detect_Stop - Detect_Start) /
     ↪  (COUNTS_PER_SECOND/1000000);
//****************************************************************************//
// Packet selection.
     XTime Select_Start, Select_Stop;
     XTime_GetTime(&Select_Start);
     int packet_idx = packetSelection(corr_out, OUT_LENGTH);

     if (packet_idx != -1){
        // printf("Packet found!\n");
         //printf("Packet start index: %d\n", packet_idx);
     }
     else
         printf("No valid packet detected.\n");



     XTime_GetTime(&Select_Stop);
     *select_time = (float)1.0 * (Select_Stop - Select_Start) /
     ↪  (COUNTS_PER_SECOND/1000000);
```

```c
//***************************************************************************//
//Downsampling
//Creating a variable to store the rx_frame
    double complex rx_frame[480];

     // Calculate end index
     int end_index = OVERSAMPLINGFACTOR * FRAME_TX_LEN + packet_idx - 1;
     //printf("\n end_index = %d \n",end_index);
    // Extract values with step size oversampling_rate_rx
    int j = 0;
    for (int i = packet_idx; i <= end_index; i += OVERSAMPLINGFACTOR) {
        rx_frame[j++] = Rx_filtered_signal[i];
    }

//***************************************************************************//
//Coarse CFO Estimation
    XTime Estimation_Start, Estimation_Stop;
    XTime_GetTime(&Estimation_Start);
    int rx_frame_len = sizeof(rx_frame) / sizeof(rx_frame[0]); //filterlength
    double complex rx_frame_after_coarse[rx_frame_len];

    coarseCFOEstimation(rx_frame,rx_frame_after_coarse,rx_frame_len);

//***************************************************************************//
// Fine CFO Estimation
     double complex rx_frame_after_fine[rx_frame_len];
    fineCFOEstimation(rx_frame_after_coarse,rx_frame_after_fine,rx_frame_len);

//***************************************************************************//
// Channel Estimation
    complex double H_est_time[N_FFT];
    double complex H_est_used_for_fft[N_FFT];
    double complex H_est[64];

    channelEstimation(rx_frame_after_fine, H_est_used_for_fft, H_est, H_est_time);
    XTime_GetTime(&Estimation_Stop);
    *estimation_time =  (float)1.0 * (Estimation_Stop - Estimation_Start) /
    ↪  (COUNTS_PER_SECOND/1000000);
//****************************************************************************//
//ONE_TAP_EQUALIZER

     complex double RX_Payload_1_Frequency[N_FFT];
     complex double RX_Payload_2_Frequency[N_FFT];
     complex double RX_Payload_1_Frequency_Equalizer[N_FFT];
     complex double RX_Payload_2_Frequency_Equalizer[N_FFT];

     oneTapEqualizer(rx_frame_after_fine, H_est, RX_Payload_1_Frequency,
      ↪  RX_Payload_2_Frequency, RX_Payload_1_Frequency_Equalizer,
      ↪  RX_Payload_2_Frequency_Equalizer);

//*****************************************************************************//
//DE_MAPPING

    double complex RX_Payload_1_no_Equalizer[48];
    double complex RX_Payload_1_no_pilot[48];
```

```
    double complex RX_Payload_2_no_Equalizer[48];
    double complex RX_Payload_2_no_pilot[48];
    demapping_RX_Payload(RX_Payload_1_Frequency, RX_Payload_1_Frequency_Equalizer,RX_Payload_1_no_Equalizer, RX_Payload_1_no_pilot);
    demapping_RX_Payload(RX_Payload_2_Frequency, RX_Payload_2_Frequency_Equalizer,RX_Payload_2_no_Equalizer, RX_Payload_2_no_pilot);

//END OF DE_MAPPING
//*********************************************************************************//
//AGC For Rx Data Payload 1
    double complex RX_Payload_1_Final[48];
    double complex RX_Payload_2_Final[48];
    Rx_Payload_AGC(RX_Payload_1_Final,RX_Payload_1_no_pilot);
    Rx_Payload_AGC(RX_Payload_2_Final,RX_Payload_2_no_pilot);
//*********************************************************************************//
//QPSK Demdoulation For Rx Data Payload 1
     int RX_Payload_1_Final_len = sizeof(RX_Payload_1_Final) / sizeof(RX_Payload_1_Final[0]);
     double  RX_Payload_1_demod[2 * RX_Payload_1_Final_len]; //96
     int RX_Payload_1_demod_len = sizeof(RX_Payload_1_demod) / sizeof(RX_Payload_1_demod[0]);

     int RX_Payload_2_Final_len = sizeof(RX_Payload_2_Final) / sizeof(RX_Payload_2_Final[0]);
     double  RX_Payload_2_demod[2 * RX_Payload_2_Final_len]; //96
     int RX_Payload_2_demod_len = sizeof(RX_Payload_2_demod) / sizeof(RX_Payload_2_demod[0]);

      demodulation_Rx_Payload(RX_Payload_1_Final_len, RX_Payload_1_Final, RX_Payload_1_demod,RX_Payload_1_demod_len);
      demodulation_Rx_Payload(RX_Payload_2_Final_len, RX_Payload_2_Final, RX_Payload_2_demod, RX_Payload_2_demod_len);
    XTime_GetTime(&Total_Rx_Stop);
    *total_rx_time = (float)1.0 * (Total_Rx_Stop - Total_Rx_Start) / (COUNTS_PER_SECOND/1000000);

    char text_1_output[12];
    char text_2_output[12];

    decode_bits_to_text(RX_Payload_1_demod, text_1_output);
   // printf("\n Decoded text_1: %s\n", text_1_output);

    decode_bits_to_text(RX_Payload_2_demod, text_2_output);
    // printf("\n Decoded text_2: %s\n", text_2_output);

    //printf("\n ****************************************************** \n");
//*******************************************************************************//
// EVM Calculation Before AGC
// Allocate an array for the concatenated error vector (size = 2*N)

    double complex error_vector[N_BITS];
   double sum_error = 0.0;
   double sum_ref = 0.0;
```

```
// Calculate error_vector = [RX_Payload_1_no_pilot, RX_Payload_2_no_pilot] - [data_payload_1_mod, data_payload_2_mod]
// Calculate sum of squared magnitudes for the error vector and for the transmitted symbols
     for (int i = 0; i < N_BITS/2; i++)
     {
     // First Payload
        error_vector[i] = RX_Payload_1_no_pilot[i] - data_payload_1_mod[i];
        sum_error += pow(cabs(error_vector[i]), 2);
        sum_ref += pow(cabs(data_payload_1_mod[i]), 2);
     // Second Payload
        error_vector[i + N_BITS/2] = RX_Payload_2_no_pilot[i] - data_payload_2_mod[i];
        sum_error += pow(cabs(error_vector[i + N_BITS/2]), 2);
        sum_ref += pow(cabs(data_payload_2_mod[i]), 2);
     }

// Calculate EVM as the RMS value of error magnitude normalized by RMS value of transmitted symbols
    double evm = sqrt(sum_error / N_BITS) / sqrt(sum_ref / N_BITS);

    // Convert EVM to dB
    double evm_dB = 20 * log10(evm);

//*******************************************************************************//
    // EVM Calculation After AGC

    double complex error_vector_AGC[N_BITS];
         sum_error = 0.0;
         sum_ref = 0.0;

// Calculate error_vector_AGC = [RX_Payload_1_Final, RX_Payload_2_Final] - [data_payload_1_mod, data_payload_2_mod]
// Calculate the sum of squared magnitudes for the error vector and the transmitted symbols
    for (int i = 0; i < N_BITS/2; i++)
    {
     // First Payload
        error_vector_AGC[i] = RX_Payload_1_Final[i] - data_payload_1_mod[i];
        sum_error += pow(cabs(error_vector_AGC[i]), 2);
        sum_ref += pow(cabs(data_payload_1_mod[i]), 2);
     // Second Payload
        error_vector_AGC[i + N_BITS/2] = RX_Payload_2_Final[i] - data_payload_2_mod[i];
        sum_error += pow(cabs(error_vector_AGC[i + N_BITS/2]), 2);
        sum_ref += pow(cabs(data_payload_2_mod[i]), 2);
    }

// Calculate EVM as the RMS error normalized by the RMS of the transmitted symbols
         double evm_AGC = sqrt(sum_error / N_BITS) / sqrt(sum_ref / N_BITS);
// Convert EVM to dB
         double evm_AGC_dB = 20 * log10(evm_AGC);

// Print results
```

```
        // printf("EVM_AGC (dB)    = %f\n", evm_AGC_dB);

//******************************************************************************//
// BER Calculation Before AGC

        // Calculate the number of error bits.
        int Error_bits = 0;
        for (int i = 0; i < 96; i++)
        {
            // Using sign_func to check if there is a difference.
            if (abs(sign_func(data_bits_1[i] - RX_Payload_1_demod[i])) == 1)
            {
                Error_bits++;
            }
        }
        for (int i = 0; i < 96; i++)
        {
            if (abs(sign_func(data_bits_2[i] - RX_Payload_2_demod[i])) == 1)
            {
                Error_bits++;
            }
        }

        // Calculate the Bit Error Rate (BER)
        double BER = (double)Error_bits / (96 + 96);

        // Adjust the size as needed.
        results[idx_snr_j].evm = evm_dB;         // Store EVM (in dB) value
        results[idx_snr_j].evm_AGC = evm_AGC_dB; // Store EVM after AGC (in dB) value
        results[idx_snr_j].ber = BER;            // Store the Bit Error Rate
        // Display the results.
       // printf("Results[%d]: evm = %lf, evm_AGC = %lf, ber = %lf\n", idx_snr_j, results[idx_snr_j].evm, results[idx_snr_j].evm_AGC, results[idx_snr_j].ber);
     }
 XTime_GetTime(&TOTAL_STOP);
 *TOTAL_EXECUTION_TIME = (float)1.0 * (TOTAL_STOP - TOTAL_START) / (COUNTS_PER_SECOND/1000000);

//******************************************************************************//
//Creating the variable array for plotting
     double evm_dB_values[NUM_SNR_VALUES];      // Equivalent to results.evm
     double evm_AGC_dB_values[NUM_SNR_VALUES];  // Equivalent to results.evm_AGC
     double ber_values[NUM_SNR_VALUES];         // Equivalent to results.ber

     // Initialize with some values (Replace with actual data)
     for (int i = 0; i < NUM_SNR_VALUES; i++) {
         evm_dB_values[i] = results[i].evm;
         evm_AGC_dB_values[i] = results[i].evm_AGC;
         ber_values[i] = results[i].ber;
     }

     // Process the arrays
     for (int i = 0; i < NUM_SNR_VALUES; i++) {
```

```c
            // Replace -inf with -40
            if (isinf(evm_AGC_dB_values[i]) && evm_AGC_dB_values[i] < 0) {
                evm_AGC_dB_values[i] = -40;
            }

            // Replace 0 with 1e-6
            if (ber_values[i] == 0) {
                ber_values[i] = 1e-6;
            }
        }

        //  // Print results for verification
        //  for (int i = 0; i < NUM_SNR_VALUES; i++) {
        //      printf("EVM: %.2f, EVM_AGC: %.2f, BER: %.6f\n",
        //             evm_dB_values[i], evm_AGC_dB_values[i], ber_values[i]);
        // }

        //****************************************************************** */
        //To copy to matlab for verification  - THE OUTPUT VALUES for EVM, BER
        printf("EVM_in_dB: \t");
        for (int i = 0; i < NUM_SNR_VALUES; i++) {
            printf("%.2f", evm_dB_values[i]);  // Print the value

            if (i < NUM_SNR_VALUES - 1) {
                printf(" , ");  // Print comma and space only if it's not the last
                ↪  element
            }
        }
        printf("\n");  // New line at the end

        printf("EVM_AGC_in_dB: \t");
        for (int i = 0; i < NUM_SNR_VALUES; i++) {
            printf("%.2f", evm_AGC_dB_values[i]);  // Print the value

            if (i < NUM_SNR_VALUES - 1) {
                printf(" , ");  // Print comma and space only if it's not the last
                ↪  element
            }
        }
        printf("\n");  // New line at the end

        printf("BER: \t");
        for (int i = 0; i < NUM_SNR_VALUES; i++) {
            printf("%.2f", ber_values[i]);  // Print the value

            if (i < NUM_SNR_VALUES - 1) {
                printf(" , ");  // Print comma and space only if it's not the last
                ↪  element
            }
        }
        printf("\n");  // New line at the end

}

int main()
```

```
{
   init_platform();
    XTime PS_Start, PS_Stop;
    float TOTAL_EXECUTION_TIME = 0;
    float total_tx_time = 0;
    float frame_tx_time = 0;
    float convolution_tx_time = 0;
    float detect_time = 0;
    float select_time = 0;
    float estimation_time = 0;
    float total_rx_time = 0;
    XTime_SetTime(0);
    XTime_GetTime(&PS_Start);

    do_main_function(&frame_tx_time, &convolution_tx_time, &total_tx_time, &detect_time, &select_time, &estimation_time, &total_rx_time, &TOTAL_EXECUTION_TIME);

    XTime_GetTime(&PS_Stop);

    float time_processor = 0;

    time_processor = (float)1.0 * (PS_Stop - PS_Start) / (COUNTS_PER_SECOND/1000000);
    printf("Execution Time for Creating the Frame_TX  : %f ms \n" , frame_tx_time/1000);
    printf("Execution Time for Convolution in Tx  : %f ms \n" , convolution_tx_time/1000);
    printf("Execution Time for Transmitter_time : %f ms \n" , total_tx_time/1000);
    printf("Execution Time for Rx_Packet_Detection  : %f ms \n" , detect_time/(1000*NUM_SNR_VALUES));
    printf("Execution Time for Rx_Packet_Selection  : %f ms \n" , select_time/(1000*NUM_SNR_VALUES));
    printf("Execution Time for RX_Channel_Estimation  : %f ms \n" , estimation_time/(1000*NUM_SNR_VALUES));
    printf("Execution Time for Receiver_Time  : %f ms \n" , total_rx_time/(1000*NUM_SNR_VALUES));
    printf("Execution Time for Total_Execution_till_EVM_BER_Calculation  : %f ms \n" , TOTAL_EXECUTION_TIME/(1000));



    printf("Execution Time for PS  : %f s \n" , time_processor/1000000);

    cleanup_platform();
    return 0;
}
```

7 Code Optimizations

7.1 text_to_binary

```
void text_to_binary(const char *text, int *binary_data) {
    for (int i = 0; i < CHAR_COUNT; i++) {
        uint8_t ch = (uint8_t) text[i];
        int base = i * 8;
        // Unrolled extraction of bits from MSB to LSB.
        binary_data[base + 0] = (ch >> 7) & 1;
        binary_data[base + 1] = (ch >> 6) & 1;
        binary_data[base + 2] = (ch >> 5) & 1;
        binary_data[base + 3] = (ch >> 4) & 1;
        binary_data[base + 4] = (ch >> 3) & 1;
        binary_data[base + 5] = (ch >> 2) & 1;
        binary_data[base + 6] = (ch >> 1) & 1;
        binary_data[base + 7] =  ch       & 1;
    }
}
```

- Loop Unrolling:

  Instead of an inner loop that extracts each bit one at a time, the inner loop is unrolled. This reduces loop overhead and may improve performance.

7.2 Short Preamble

- Reduced 3 `for` loops into a single one for setting up the short preamble slot frequency array
- Reduced a for loop by using modulus operator for repeating the first 16 values ten values

```
// Function to create the short preamble
void create_short_preamble(complex double *Short_preamble) {
    // Virtual subcarriers (padding zeros)
    complex double virtual_subcarrier[11] = {0};
    // Constructing the frequency-domain preamble (64 elements)
    complex double Short_preamble_slot_Frequency[N_FFT] = {0};
    for (int i = 0; i < 53; i++)
        Short_preamble_slot_Frequency[i + 6] = S_k[i] * sqrt(13.0 / 6);
    // Perform IFFT shift
    fftshift(Short_preamble_slot_Frequency, N_FFT);
    // Time-domain representation of the short preamble
    complex double Short_preamble_slot_Time[N_FFT];
    ifft(Short_preamble_slot_Frequency, Short_preamble_slot_Time, N_FFT);
 // Repeat first 16 values of the time-domain representation 10 times
    for (int i = 0; i < 160; i++)
        Short_preamble[i] = Short_preamble_slot_Time[i % 16];
}
}
```

7.3 Long Preamble

- `memcpy` is faster than manual loops as it copies `L_k` efficiently in one go.
- Reduced a `for` loop

```
void create_long_preamble(complex double *Long_preamble) {
    // Frequency-domain preamble (64 elements)
    complex double Long_preamble_slot_Frequency[N_FFT] = {0};
    // Initializes with zero // Copy L_k into appropriate positions
    memcpy(&Long_preamble_slot_Frequency[6], L_k, 53 * sizeof(complex double));
    // Perform IFFT shift directly before calling IFFT
    fftshift(Long_preamble_slot_Frequency, N_FFT);
    // Time-domain representation
    complex double Long_preamble_slot_Time[N_FFT];
    ifft(Long_preamble_slot_Frequency, Long_preamble_slot_Time, N_FFT);

    for (int i = 0; i < 32; i++) {
       Long_preamble[i] = Long_preamble_slot_Time[i + 32];  // First 32 elements
        ↪  from shifted IFFT
    }
    for (int i = 0; i < 64; i++) {
        Long_preamble[i + 32] = Long_preamble_slot_Time[i];  // Main IFFT output
        Long_preamble[i + 96] = Long_preamble_slot_Time[i];  // Duplicate main IFFT
         ↪  output
    }
}
```

7.4 FFT Codes

```
// Global cache for twiddle factors (for FFT of size N)
static complex double *twiddleCache = NULL;
static int cacheSize = 0;

// Initialize (or reinitialize) the twiddle factor cache for FFT of size N
void initTwiddleCache(int N) {
    if (cacheSize == N && twiddleCache != NULL)
        return;  // Already computed for this size
    if (twiddleCache != NULL)
        free(twiddleCache);
    cacheSize = N;
    // We only need N/2 twiddle factors for an FFT of size N.
    twiddleCache = malloc(sizeof(complex double) * (N / 2));
    for (int k = 0; k < N / 2; k++) {
        twiddleCache[k] = cexp(-I * 2 * PI * k / N);
    }
}

// Bit reversal function for reordering the input array
unsigned int bitReverse(unsigned int x, int log2n) {
    unsigned int n = 0;
    for (int i = 0; i < log2n; i++) {
        n = (n << 1) | (x & 1);
```

```
        x >>= 1;
    }
    return n;
}

// Iterative FFT using the cached twiddle factors (DP approach)
void fft(complex double *x, complex double *X, int N) {
    initTwiddleCache(N);

    // Compute log2(N)
    int log2N = 0;
    for (int n = N; n > 1; n >>= 1)
        log2N++;

    // Bit-reversal permutation: reorder x into X
    for (unsigned int i = 0; i < (unsigned int)N; i++) {
        unsigned int j = bitReverse(i, log2N);
        X[j] = x[i];
    }

    // Danielson-Lanczos section: perform butterfly computations in stages.
    for (int s = 1; s <= log2N; s++) {
        int m = 1 << s;      // m = 2^s, the current sub-DFT size
        int m2 = m >> 1;     // m/2, the number of butterflies per sub-DFT
        for (int k = 0; k < N; k += m) {
            for (int j = 0; j < m2; j++) {
                // Use the cached twiddle factor.
                // The index is scaled: (N * j) / m gives the appropriate index.
                complex double t = twiddleCache[(N * j) / m] * X[k + j + m2];
                complex double u = X[k + j];
                X[k + j] = u + t;
                X[k + j + m2] = u - t;
            }
        }
    }
}

// Optimized IFFT using the conjugated FFT approach with dynamic programming
// (Here, fft_dp reuses cached twiddle factors to avoid recomputation.)
void ifft(complex double *X, complex double *x, int N) {
    // Allocate temporary arrays for the conjugated input and FFT result
    complex double *X_conj = malloc(sizeof(complex double) * N);
    complex double *temp   = malloc(sizeof(complex double) * N);
    if (!X_conj || !temp) {
        free(X_conj);
        free(temp);
        return;  // Allocation failed
    }

    // Conjugate the input array X
    for (int i = 0; i < N; i++) {
        X_conj[i] = conj(X[i]);
    }

    // Compute FFT on the conjugated input using our DP-optimized FFT
```

```
    fft(X_conj, temp, N);

    // Conjugate the result and scale by 1/N to obtain the IFFT
    for (int i = 0; i < N; i++) {
        x[i] = conj(temp[i]) / N;
    }

    free(X_conj);
    free(temp);
}

// Function to perform FFT shift (reordering the frequency bins)
void fftshift(complex double *X, int N) {
    int mid = N / 2;
    for (int i = 0; i < mid; i++) {
        complex double temp = X[i];
        X[i] = X[i + mid];
        X[i + mid] = temp;
    }
}
```

- Twiddle Factor Precomputation:

  This step is O(N) (for N/2 factors), but since it's cached and reused, it's computed only once per FFT size.

- FFT Computation:

  The iterative FFT uses log(N) stages, with each stage processing O(N) operations (butterfly computations), leading to an overall O(N log N) complexity.

- Optimized IFFT:

  The conjugated FFT approach involves conjugating the input (O(N)), performing an FFT (O(N log N)), and then conjugating and scaling the output (O(N)), so the overall complexity remains O(N log N).

7.5 Tx_frame

```
// Function to create transmitted frame
void create_frame_tx(double complex *Short_preamble, double complex *Long_preamble,
    double complex *data_1_TX_payload, double complex *data_2_TX_payload,
    double complex *Frame_Tx) {
// Use memcpy to copy memory blocks
memcpy(Frame_Tx, Short_preamble, 160 * sizeof(double complex));          //
↪  Copy Short Preamble
memcpy(Frame_Tx + 160, Long_preamble, 160 * sizeof(double complex));     //
↪  Copy Long Preamble
memcpy(Frame_Tx + 320, data_1_TX_payload, 80 * sizeof(double complex));  //
↪  Copy Data 1 Payload
memcpy(Frame_Tx + 400, data_2_TX_payload, 80 * sizeof(double complex));  //
↪  Copy Data 2 Payload
}
```

7.6 QPSK Modulation

```
// Function for QPSK Modulation
void qpsk_modulation(int *data_bits, complex double *modulated_symbols, int size) {
    static const complex double QPSK_LUT[4] = {
        (1 + I) * INV_SQRT2,   // 00 -> (1 + j)/2
        (-1 + I) * INV_SQRT2,  // 01 -> (-1 + j)/2
        (-1 - I) * INV_SQRT2,  // 10 -> (-1 - j)/2
        (1 - I) * INV_SQRT2    // 11 -> (1 - j)/2
    };
    for (int i = 0; i < size / 2; i++) {
        int index = (data_bits[2 * i] << 1) | data_bits[2 * i + 1];  // Convert 2
        ↪  bits to index (00 → 0, 01 → 1, 10 → 2, 11 → 3)
        modulated_symbols[i] = QPSK_LUT[index];  // Direct LUT lookup
    }
}
```

7.7 Packet Detection

```
void compute_normalized_correlation(const double complex Rx_signal[], int len_Rx,
                                    int delay_param, int window_length,
                                    double corr_out[], int out_length) {
    double complex sum_corr = 0.0 + 0.0 * I;
    double sum_peak = 0.0;
    int n, k;

    // Compute the first window
    for (k = 0; k < window_length; k++) {
        double complex sample1 = Rx_signal[k];
        double complex sample2 = Rx_signal[k + delay_param];
        sum_corr += sample1 * sample2;
        double sample2_abs = cabs(sample2);
        sum_peak += sample2_abs * sample2_abs; // replacing pow(cabs(), 2)
    }
    if (sum_peak != 0)
        corr_out[0] = (cabs(sum_corr) * cabs(sum_corr)) / (sum_peak * sum_peak);
    else
        corr_out[0] = 0.0;

    // Slide the window and update sums incrementally
    for (n = 1; n < out_length; n++) {
        // Remove the contribution of the element that is leaving the window
        double complex old_sample1 = Rx_signal[n - 1];
        double complex old_sample2 = Rx_signal[n - 1 + delay_param];
        sum_corr -= old_sample1 * old_sample2;
        double old_sample2_abs = cabs(old_sample2);
        sum_peak -= old_sample2_abs * old_sample2_abs;

        // Add the contribution of the new element entering the window
        double complex new_sample1 = Rx_signal[n + window_length - 1];
        double complex new_sample2 = Rx_signal[n + window_length - 1 +
        ↪  delay_param];
        sum_corr += new_sample1 * new_sample2;
        double new_sample2_abs = cabs(new_sample2);
```

```
        sum_peak += new_sample2_abs * new_sample2_abs;

        // Compute normalized correlation for the current window
        if (sum_peak != 0)
            corr_out[n] = (cabs(sum_corr) * cabs(sum_corr)) / (sum_peak *
            ↪  sum_peak);
        else
            corr_out[n] = 0.0;
    }
}
```

- Previously, we recompute the entire sum for each window, leading to an O(out_length × window_length) complexity. Since the windows overlap, we can update the sums incrementally using a sliding window approach.
- In this method, we calculate the sum for the first window, and then for each subsequent window, subtract the contribution of the sample leaving the window and add the new sample's contribution.
- This reduces the per-window update to O(1), making the overall time complexity O(window_length + out_length).
- Sliding Window Update:

  By updating the sums incrementally, we avoid recomputing the entire sum for every window.
- Replacing `pow(cabs(sample2), 2)`:

  Directly computing the squared magnitude using multiplication (i.e., `(sample2_abs * sample2_abs)` is typically faster than calling `pow`.

7.8 Packet Selection

```
int packetSelection(const double corr_out[], int corr_out_length)
{
    int packet_candidates[OUT_LENGTH];
    int num_candidates = 0;
    int last_packet_index = -300; // Initialize so the first valid index always
    ↪  qualifies

    // Collect packet candidate indices where corr_out > PACKET_THRESHOLD and gap >
    ↪  300
    for (int i = 0; i < corr_out_length; i++) {
        if (corr_out[i] > PACKET_THRESHOLD) {
            if (i - last_packet_index > 300) {  // Ensure sufficient gap between
            ↪  candidates
                packet_candidates[num_candidates++] = i;
                last_packet_index = i;
                if (num_candidates >= OUT_LENGTH)
                    break;
            }
        }
    }
```

```
    // Check each candidate: if the sample at candidate + THRESHOLD_LENGTH exceeds
    ↪  the threshold,
    // return the candidate adjusted by LENGTH_RRC_RX (assumed here as 11).
    for (int i = 0; i < num_candidates; i++) {
        int packet_candidate = packet_candidates[i];
        int check_index = packet_candidate + THRESHOLD_LENGTH;
        if (check_index < corr_out_length && corr_out[check_index] >
        ↪  PACKET_THRESHOLD) {
            return packet_candidate + 11;
        }
    }
    return -1;
}
```

- packet_candidates and num_candidates:

  These variables store and count the indices in corr_out that exceed the threshold and are separated by at least 300 samples from the previous candidate.

Appendix

Listing 1: FFT Definitions

```c
// Function to perform IFFT using Cooley-Tukey Algorithm
void ifft(complex double *X, complex double *x, int N) {
    // Take complex conjugate of input
    for (int i = 0; i < N; i++) {
        X[i] = conj(X[i]);
    }

    // Perform FFT on conjugated input
    complex double *temp_output = malloc(N * sizeof(complex double));
    fft(X, temp_output, N);

    // Take complex conjugate of result and normalize
    for (int i = 0; i < N; i++) {
        x[i] = conj(temp_output[i]) / N;
    }

    // Free allocated memory
    free(temp_output);
}

// Function to perform FFT shift (reordering the frequency bins)
void fftshift(complex double *X, int N) {
    int mid = N / 2;
    for (int i = 0; i < mid; i++) {
        complex double temp = X[i];
        X[i] = X[i + mid];
        X[i + mid] = temp;
    }
}

// Function to perform FFT using Cooley-Tukey Algorithm
void fft(complex double *x, complex double *X, int N) {
    // Base case: If N = 1, just copy input to output
    if (N == 1) {
        X[0] = x[0];
        return;
    }

    // Split arrays into even and odd indexed elements
    complex double *even = malloc(N / 2 * sizeof(complex double));
    complex double *odd = malloc(N / 2 * sizeof(complex double));
    complex double *X_even = malloc(N / 2 * sizeof(complex double));
    complex double *X_odd = malloc(N / 2 * sizeof(complex double));

    for (int i = 0; i < N / 2; i++) {
        even[i] = x[2 * i];      // Even-indexed samples
        odd[i] = x[2 * i + 1];  // Odd-indexed samples
    }

    // Recursive calls for FFT on even and odd indexed elements
    fft(even, X_even, N / 2);
```

```
    fft(odd, X_odd, N / 2);

    // Compute FFT output using results of smaller FFTs
    for (int k = 0; k < N / 2; k++) {
        complex double twiddle = cexp(-I * 2 * PI * k / N) * X_odd[k];
        X[k] = X_even[k] + twiddle;
        X[k + N / 2] = X_even[k] - twiddle;
    }

    // Free allocated memory
    free(even);
    free(odd);
    free(X_even);
    free(X_odd);
}
```

Listing 2: Payload Definitions

```
//Function to generate bits from text data
void text_to_binary(const char *text, int *binary_data) {
    for (int i = 0; i < CHAR_COUNT; i++) {
        uint8_t ch = text[i];
        for (int j = 7; j >= 0; j--) {  // Extract bits from MSB to LSB
            binary_data[i * 8 + (7 - j)] = (ch >> j) & 1;
        }
    }
}

// Function to perform QPSK modulation
void qpsk_modulation(int *data_bits, complex double *modulated_symbols, int size) {
    for (int i = 0; i < size / 2; i++) {
        int bit1 = data_bits[2 * i];
        int bit2 = data_bits[2 * i + 1];

        if (bit1 == 0 && bit2 == 0)
            modulated_symbols[i] = (1 + I) / sqrt(2);
        else if (bit1 == 0 && bit2 == 1)
            modulated_symbols[i] = (-1 + I) / sqrt(2);
        else if (bit1 == 1 && bit2 == 0)
            modulated_symbols[i] = (-1 - I) / sqrt(2);
        else
            modulated_symbols[i] = (1 - I) / sqrt(2);
    }
}

// Function to construct the frame
void construct_frame(complex double *payload, complex double *frame,
    complex double *virtual_subcarrier, complex double *pilot) {
memcpy(frame, virtual_subcarrier, 6 * sizeof(complex double));// Virtual
↪  subcarriers [0-5]
memcpy(frame + 6, payload, 5 * sizeof(complex double));       // Data [6-10]
frame[11] = pilot[0];                                        // Pilot at index 11
memcpy(frame + 12, payload + 5, 13 * sizeof(complex double)); // Data [12-24]
frame[25] = pilot[1];                                         // Pilot at index 25
memcpy(frame + 26, payload + 18, 6 * sizeof(complex double)); // Data [26-31]
```

```
frame[32] = 0;                                              // Null subcarrier at
↪  index 32
memcpy(frame + 33, payload + 24, 6 * sizeof(complex double)); // Data [33-38]
frame[39] = pilot[2];                                       // Pilot at index 39
memcpy(frame + 40, payload + 30, 13 * sizeof(complex double));// Data [40-52]
frame[53] = pilot[3];                                       // Pilot at index 53
memcpy(frame + 54, payload + 43, 5 * sizeof(complex double)); // Data [54-58]
memcpy(frame + 59, virtual_subcarrier + 6, 5 * sizeof(complex double)); // Virtual
↪  subcarriers [59-63]
}
```

Listing 3: Correlation

```
void compute_normalized_correlation(const double complex Rx_signal[],
    int len_Rx,
    int delay_param,
    int window_length,
    double corr_out[],
    int out_length)
{
    int n, k;

    // Compute cross-correlation and normalized correlation in a single pass
    for (n = 0; n < out_length; n++) {
        double complex sum_corr = 0.0 + 0.0 * I;
        double sum_peak = 0.0;

        for (k = 0; k < window_length; k++) {
            double complex sample1 = Rx_signal[n + k];
            double complex sample2 = Rx_signal[n + k + delay_param];

            sum_corr += sample1 * sample2;  // Cross-correlation
            sum_peak += cabs(sample2) * cabs(sample2);  // Power sum
        }

        // Compute normalized correlation directly
        double mag_corr_sq = cabs(sum_corr) * cabs(sum_corr);
        corr_out[n] = (sum_peak > 0.0) ? (mag_corr_sq / (sum_peak * sum_peak)) :
        ↪  0.0;
    }
}
```

Listing 4: Data Declarations

```c
 #ifndef SRC_DATA_H_
 #define SRC_DATA_H_
 #include <complex.h>
 #define N_FFT 64 // FFT Size
 #define PI 3.14159265358979323846
 #define FC_HZ 5000000000 // Carrier frequency (5 GHz)
 #define FS_HZ 20000000   // Sampling frequency (20 MHz)
 #define TS_SEC (1.0 / FS_HZ) // Time period (50 ns)
 #define N_FFT 64        // FFT Size
 #define N_BITS 96       // Number of bits in one frame
 #define CHAR_COUNT (N_BITS / 8) //Number of characters that fit
 #define M 4             // QPSK modulation
 #define FRAME_TX_LEN 480    // Define the total length of Frame_Tx
 #define OVERSAMPLINGFACTOR 2 //Oversampling factor
 #define FRAME_TX_OVERSAMP_LENGTH (FRAME_TX_LEN  * OVERSAMPLINGFACTOR)
 #define NUM_SNR_VALUES 10
 #define NP_PACKETS_CAPTURE 3000 // Number of packets to capture
 #define M_SQRT2 1.41421356237309504880
#define INV_SQRT2 0.7071067811865475  // Precomputed 1/sqrt(2)

// Inputs for Packet Detection:
//   Rx_signal: input array of complex samples (length = LEN_RX)
//   len_Rx: length of Rx_signal (fixed as LEN_RX)
//   delay_param: delay applied for correlation
//   window_length: length of the window over which correlation is computed
//   out_length: number of windows (should be LEN_RX - delay_param + 1 - window_length)
#define LEN_RX 3000         // Length of Rx_signal array
#define DELAY_PARAM 16      // Delay parameter for correlation
#define WINDOW_LENGTH 32    // Correlation window length
#define OUT_LENGTH (LEN_RX - DELAY_PARAM + 1 - WINDOW_LENGTH) // No of output windows

#define SHORT_PREAMBLE_SLOT_LENGTH 16
#define PACKET_THRESHOLD 0.75
#define THRESHOLD_LENGTH 230

//**Defined the array elements in data.c
//// Short Preamble sequence (53 elements) **//
extern const complex double S_k[53];  // Declare the array using extern;
///Defined the array elements in data.c
//// Long Preamble sequence (53 elements)
extern const complex double L_k[53]; //Declare the array using extern;
extern const char text_1[12];  // Use 'extern' to declare but not define
extern const char text_2[12];
// Defining  the Tx RRC filter coefficients
extern const double rrc_filter_tx[21];
// Defining  the Rx RRC filter coefficients
extern const double rrc_filter_rx[21];

 #endif /* SRC_DATA_H_ */
```

```c
#include "data.h"
#include <complex.h>



const char text_1[12] = "HELLO_EMBED!"; //Const 12 char message for 96 bits
const char text_2[12] = "EMBEDDED_SYS"; //Const 12 char message for 96 bits

const complex double S_k[53] = {
    0, 0, 1 + I, 0, 0, 0, -1 - I, 0, 0, 0, 1 + I, 0, 0, 0, -1 - I, 0, 0, 0, -1 - I,
    ↪  0, 0, 0,
    1 + I, 0, 0, 0, 0, 0, 0, 0, -1 - I, 0, 0, 0, -1 - I, 0, 0, 0, 1 + I, 0, 0, 0, 1
    ↪  + I, 0, 0,
    0, 1 + I, 0, 0, 0, 1 + I, 0, 0
};

const complex double  L_k[53] = {
    1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1, 1,
    ↪  1, 1,
    0, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, -1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1,
    ↪  -1, 1, 1, 1, 1
};

const double rrc_filter_tx[21] = {
    -0.000454720514876, 0.003536895555750, -0.007145608090912, 0.007579061905178,
    0.002143682427274, -0.010610686667250, 0.030011553981831, -0.053053433336248,
    -0.075028884954579, 0.409168714634052, 0.803738600397980, 0.409168714634052,
    -0.075028884954579, -0.053053433336248, 0.030011553981831, -0.010610686667250,
    0.002143682427274, 0.007579061905178, -0.007145608090912, 0.003536895555750,
    -0.000454720514876
};

const double rrc_filter_rx[21] = {
    -0.000454720514876, 0.003536895555750, -0.007145608090912, 0.007579061905178,
    0.002143682427274, -0.010610686667250, 0.030011553981831, -0.053053433336248,
    -0.075028884954579, 0.409168714634052, 0.803738600397980, 0.409168714634052,
    -0.075028884954579, -0.053053433336248, 0.030011553981831, -0.010610686667250,
    0.002143682427274, 0.007579061905178, -0.007145608090912, 0.003536895555750,
    -0.000454720514876
};
```