ECE 230B: Lab 3

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as signal
from ece230b import *
```

In [71]:
```
# pluto_token = '7d1-EddvbnU' # 1:30-2:00am
# pluto_token = 'NS8uqBzWDLU' # 2:00-2:30am
# pluto_token = 'uJ_6FnhzugM' # 2:30-3:00am

# pluto_token = 'U-YGolM3_bU' # 3:30-4:00pm
# pluto_token = 'lfz_DVf4PxU' # 4:00-4:30pm
# pluto_token = '7U9uolsF83o' # 4:30-5:00pm
# pluto_token = 'UIOHCyG4vtg' # 5:00-5:30pm
# pluto_token = 'Bz12my4ENdY' # 6:00-6:30pm

# pluto_token = 'OQsj2cfGEA0' # 12:00-12:30am
# pluto_token = '7YxZhZw4CgA' # 12:30-1:00am
# pluto_token = 'pdKsufv4mjU' # 1:00-1:30am
# pluto_token = 'FC3fKrbrcLE' # 2:00-2:30am
# pluto_token = 'UWBvlEJtuS4' # 2:30-3:00am
# pluto_token = 'ytrJsHht5iQ' # 3:00-3:30am
pluto_token = 'MMPTaXKoIV4' # 3:30-4:00am
# pluto_token = None

%matplotlib inline
```

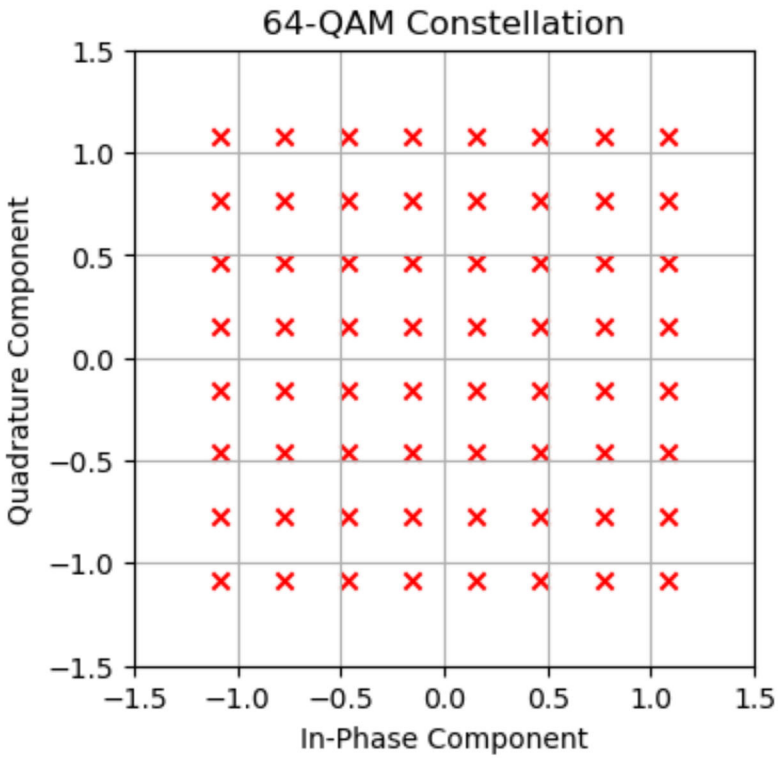
Part I: The Transmitter

Begin by generating `N=1000` random `M`-QAM symbols, drawn uniformly from a constellation with unit average energy per symbol, `E_s=1`. Let's start with 64-QAM. Optionally, insert `N/10` zeros before or after your sequence of symbols. This will make it a bit easier to recognize your signal in the time domain. Then, perform pulse shaping at the transmitter using a root raised cosine (RRC) filter at a symbol rate of `1/T=10^5` symbols per second, meaning symbols are sent every 10 microseconds. Use a sampling rate of 1MSPS, and thus the number of samples should be 10. Use a rolloff factor of `beta=0.5`. Use a cyclic transmit buffer, and set the transmit gain to `-25dB`. **Plot** the real and imaginary parts of the transmit signal, zooming in so your pulse-shaped symbols can be observed clearly. Transmit this pulse-shaped waveform from the Pluto at a carrier frequency of your choice between 905-925 MHz using `sdr.tx()`.

Symbol Generation

In [3]:
```
# Generate 1000 random 64-QAM symbols
N=1000
M=64
```

```
symbols, constellation = gen_rand_qam_symbols(N, M)
# Insert zeros at beginning of symbols
# zeros_symbols = np.concatenate((np.zeros(int(N/10), dtype=complex), symbols))
zeros_symbols = symbols

# Visualize constellation
%matplotlib inline
plt.figure(figsize=(4, 4))
plt.scatter(constellation.real, constellation.imag, c='red', marker='x')
plt.title(f"{M}-QAM Constellation")
plt.xlabel("In-Phase Component")
plt.ylabel("Quadrature Component")
plt.grid(True)
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()
```

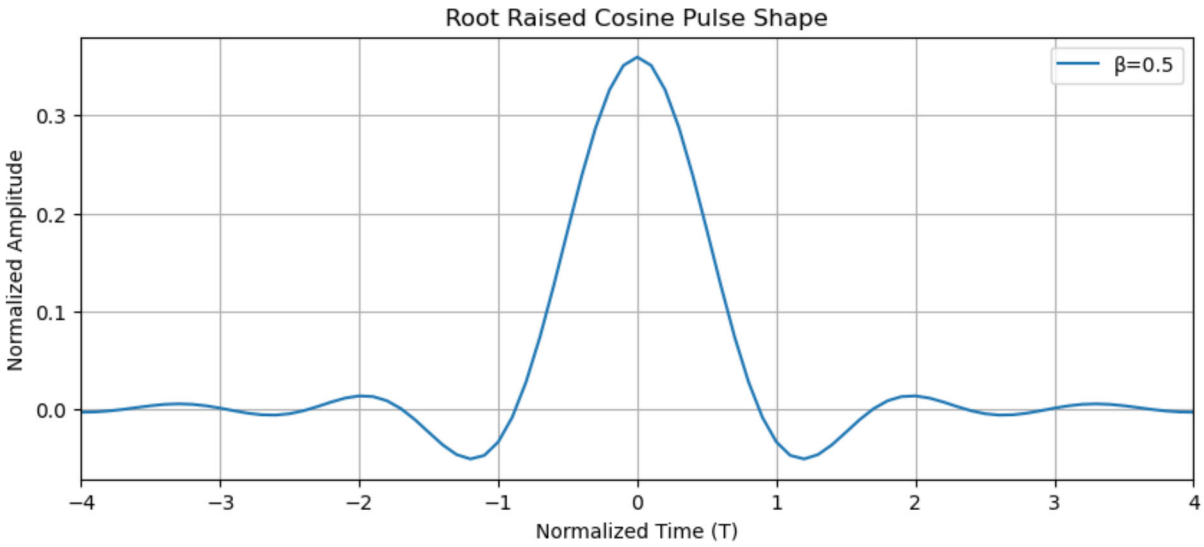


TX Signal Generation

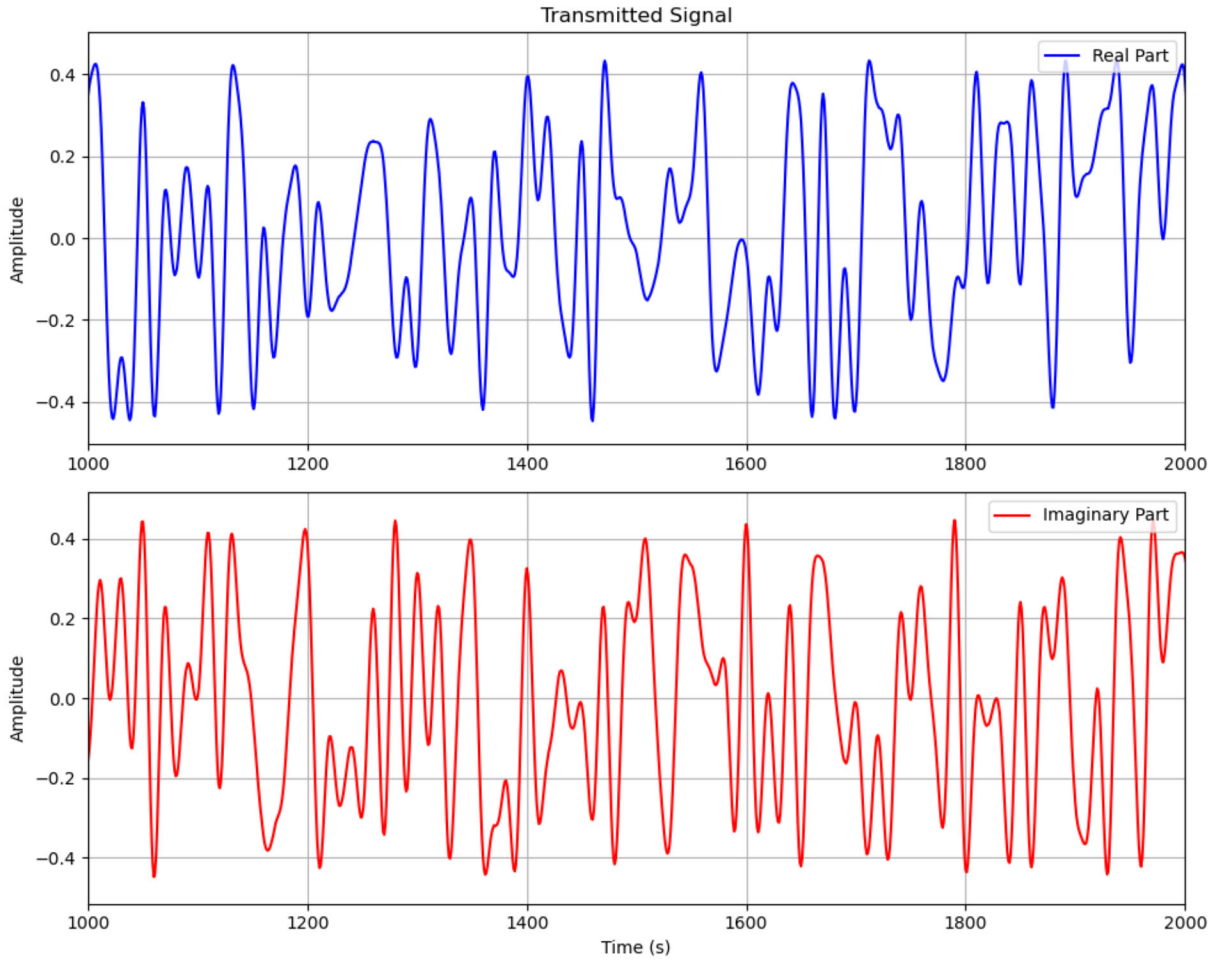
In [4]:
```
# Perform pulse shaping with RRC filter
span = 8 # Number of symbols to span
sps = 10 # Samples per symbol
beta = 0.5 # Roll-off factor

t_filter, g_tx = get_rrc_pulse(beta, span, sps) # normalalized to unit norm
print(f"norm of g_tx: {np.sum(np.abs(g_tx)**2)}")  # Check energy of the pulse
# t_filter, g_tx = filters.rrcosfilter(span*sps, beta, 1, sps)

# Visualize pulse shape
plt.figure(figsize=(10, 4))
plt.plot(t_filter, g_tx, label=f'β={beta}')
plt.title("Root Raised Cosine Pulse Shape")
```

```
plt.xlabel("Normalized Time (T)")
plt.ylabel("Normalized Amplitude")
plt.xlim(-span/2, span/2)
plt.grid(True)
plt.legend()
plt.show()
```

```
norm of g_tx: 1.0
```



In [5]:
```
t, pulse_train = create_pulse_train(zeros_symbols, sps)

tx_signal = np.convolve(g_tx, pulse_train, mode='same')

# Visualize tx signal
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Transmitted Signal")
plt.plot(t, np.real(tx_signal), label='Real Part', color='blue')
plt.xlim(N/10*sps, 2*N/10*sps)
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(t, np.imag(tx_signal), label='Imaginary Part', color='red')
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.xlim(N/10*sps, 2*N/10*sps)
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```



Pluto SDR Setup

In [6]:
```
# Pluto SDR Setup
from remoteRF.drivers.adalm_pluto import *
# -------------------------------------------------------------
# Digital communication system parameters.
# -------------------------------------------------------------
fs = 1e6      # baseband sampling rate (samples per second)
ts = 1 / fs   # baseband sampling period (seconds per sample)
sps = 10      # samples per data symbol
T = ts * sps # time between data symbols (seconds per symbol)

# -------------------------------------------------------------
# Pluto system parameters.
# -------------------------------------------------------------
sample_rate = fs                # sampling rate, between ~600e3 and 61e6
tx_carrier_freq_Hz = 915e6      # transmit carrier frequency, between 325 MHz to 3.
rx_carrier_freq_Hz = 915e6      # receive carrier frequency, between 325 MHz to 3.8
tx_rf_bw_Hz = sample_rate * 1   # transmitter's RF bandwidth, between 200 kHz and 5
rx_rf_bw_Hz = sample_rate * 1   # receiver's RF bandwidth, between 200 kHz and 56 M
tx_gain_dB = -25                # transmit gain (in dB), beteween -89.75 to 0 dB wi
rx_gain_dB = 40                 # receive gain (in dB), beteween 0 to 74.5 dB (only
rx_agc_mode = 'manual'          # receiver's AGC mode: 'manual', 'slow_attack', or
rx_buffer_size = 500e3          # receiver's buffer size (in samples), length of da
tx_cyclic_buffer = True         # cyclic nature of transmitter's buffer (True -> co
```

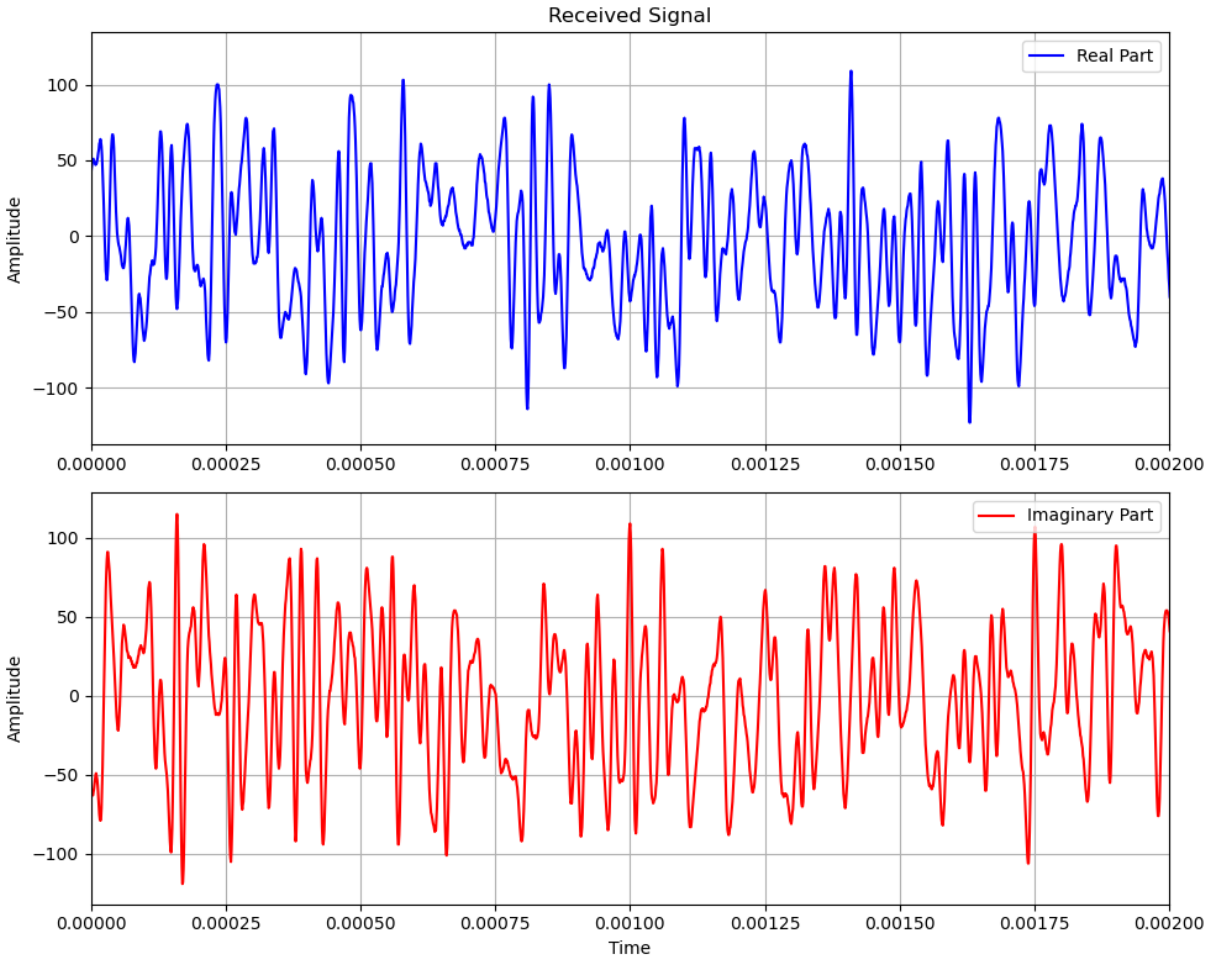
```
# ------------------------------------------------------------
# Initialize Pluto object using issued token.
# ------------------------------------------------------------
if pluto_token is not None:
    sdr = adi.Pluto(token=pluto_token) # create Pluto object
    sdr.sample_rate = int(sample_rate)   # set baseband sampling rate of Pluto

# ------------------------------------------------------------
# Setup Pluto's transmitter.
# ------------------------------------------------------------
    sdr.tx_destroy_buffer()                  # reset transmit data buffer to be sa
    sdr.tx_rf_bandwidth = int(tx_rf_bw_Hz)   # set transmitter RF bandwidth
    sdr.tx_lo = int(tx_carrier_freq_Hz)      # set carrier frequency for transmiss
    sdr.tx_hardwaregain_chan0 = tx_gain_dB   # set the transmit gain
    sdr.tx_cyclic_buffer = tx_cyclic_buffer  # set the cyclic nature of the transm

# ------------------------------------------------------------
# Setup Pluto's receiver.
# ------------------------------------------------------------
    sdr.rx_destroy_buffer()                  # reset receive data buffer to be saf
    sdr.rx_lo = int(rx_carrier_freq_Hz)      # set carrier frequency for reception
    sdr.rx_rf_bandwidth = int(sample_rate)   # set receiver RF bandwidth
    sdr.rx_buffer_size = int(rx_buffer_size) # set buffer size of receiver
    sdr.gain_control_mode_chan0 = rx_agc_mode # set gain control mode
    sdr.rx_hardwaregain_chan0 = rx_gain_dB   # set gain of receiver
```

Part II: The Receiver

Set the receiver's carrier frequency to that matching the transmitter. Set the receiver's gain control to `"manual"`, and use a gain of 40 dB. Set the receive buffer size to `500*10^3` samples. Take a receive capture with the Pluto using `sdr.rx()`. **Plot** the real and imaginary parts of the received signal, zooming in so the received pulse-shaped waveform can be observed closely.

In [7]:
```
# Transmit and receive the signal

if pluto_token is not None:
    # Transmit signal
    tx_scaled = tx_signal / np.max(np.abs(tx_signal)) * 2**14 # Pluto expects TX sa
    sdr.tx_destroy_buffer()  # reset transmit data buffer to be safe
    sdr.tx(tx_scaled)

    # Receive signal
    sdr.rx_destroy_buffer()
    rx_signal = sdr.rx() # / np.max(np.abs(sdr.rx()))

    t = np.arange(rx_buffer_size) / sample_rate  # time vector for TX signal

else:
    print("No Pluto token provided, simulating transmission and reception.")
    N0 = 0.001
    noise = np.sqrt(N0/2) * (np.random.randn(len(tx_signal)) + 1j * np.random.randn
    rx_signal = tx_signal + noise
```

```
    t = np.arange(len(zeros_symbols) * sps) / sample_rate  # time vector for TX sig

# Visualize received signal
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Received Signal")
plt.plot(t, np.real(rx_signal), label='Real Part', color='blue')
plt.ylabel("Amplitude")
plt.xlim(0, t[int(2*N/10*sps)])
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(t, np.imag(rx_signal), label='Imaginary Part', color='red')
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.xlim(0, t[int(2*N/10*sps)])
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```



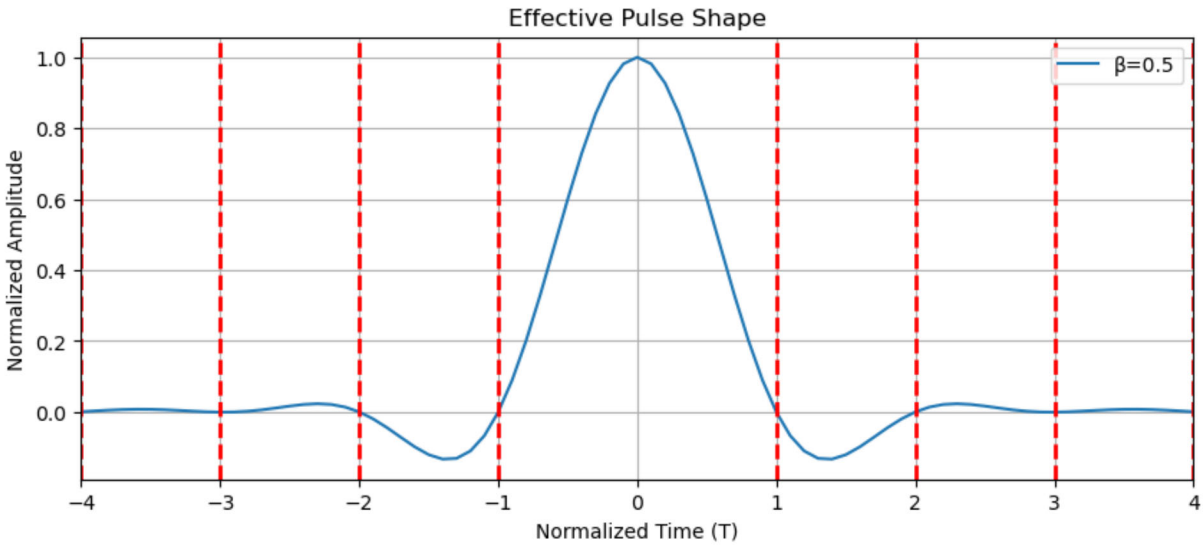
Part III: Matched Filter

Pass the received signal through a filter matched to the transmitter's pulse shape. **Plot** the

impulse response of the *effective* pulse shape to confirm it indeed satisfies the Nyquist ISI criterion. **Plot** the real and imaginary parts of the matched filter output.

Generate Matched Filter

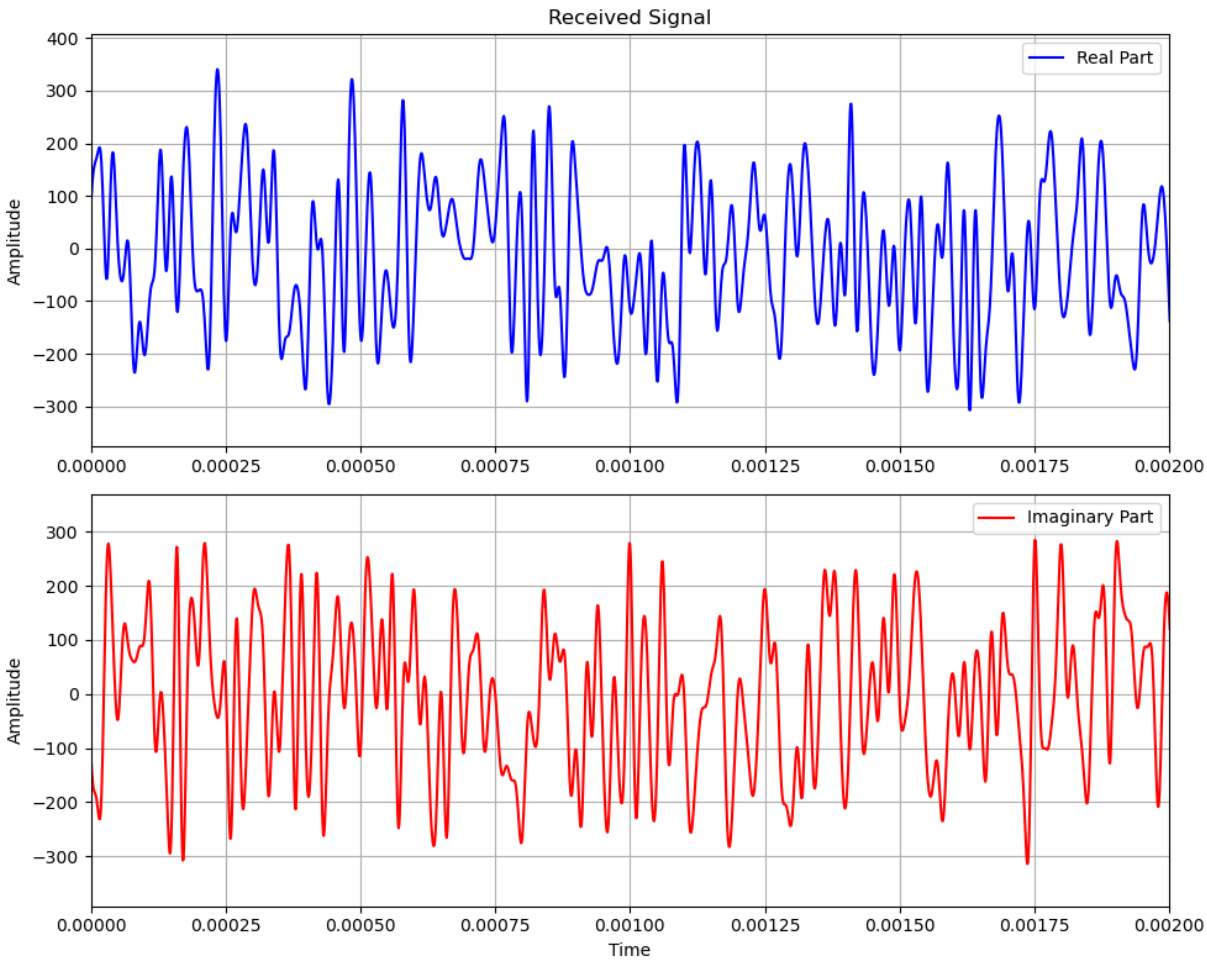
In [8]:
```
# Generate receiver matched filter
g_rx = np.conj(g_tx[::-1])

# Visualize effective pulse shape
effective_pulse = np.convolve(g_tx, g_rx, mode='same')
plt.figure(figsize=(10, 4))
plt.plot(t_filter, effective_pulse, label=f'β={beta}')
for k in range(int(np.ceil(t_filter[0])), int(np.floor(t_filter[-1])) + 1):
    if k != 0: plt.axvline(k, color='red', lw=2, ls='--')
plt.title("Effective Pulse Shape")
plt.xlabel("Normalized Time (T)")
plt.ylabel("Normalized Amplitude")
plt.xlim(-span/2, span/2)
plt.grid(True)
plt.legend(loc='upper right')
plt.show()
```



Apply Matched Filter

In [9]:
```
# Pass received signal through matched filter
rx_signal_mf = np.convolve(rx_signal, g_rx, mode='same')

# Visualize matched filter output
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Received Signal")
plt.plot(t, np.real(rx_signal_mf), label='Real Part', color='blue')
plt.ylabel("Amplitude")
plt.xlim(0, t[int(2*N/10*sps)])
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
```

```
plt.plot(t, np.imag(rx_signal_mf), label='Imaginary Part', color='red')
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.xlim(0, t[int(2*N/10*sps)])
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```



Part IV: Frame Synchronization

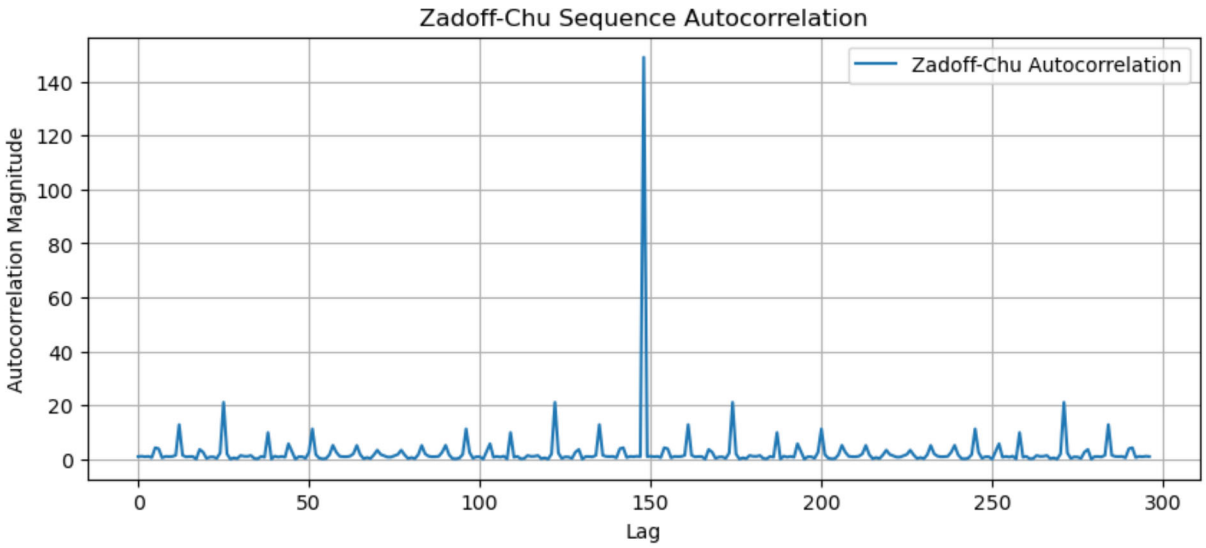
You'll notice your received signal is not perfectly time-aligned with the transmit signal. In other words, the received copy of the transmit signal has some time offset that needs to be accounted for at the receiver. Correcting for this offset is the problem of timing synchronization, as discussed in class. You'll also notice that you have received multiple copies of your transmitted signal, due to the size of your receive buffer relative to the length of the signal (which was cyclically transmitted). We only want to extract one copy of the transmitted signal. Prepend to your transmitted symbols a Zadoff-Chu sequence to help us locate the transmitted signal. Assume that this sequence is known to the receiver. Given our received signal is upsampled by a factor of 10 samples per symbol, the receiver can either correlate with the Zadoff-Chu symbols (one out of every ten samples) or correlate with a pulse-shaped version of those symbols. Perform frame synchronization using one (or both)

of these methods and extract a single copy of the transmitted signal. **Plot** the magnitude of the cross-correlation output during frame synchronization.

Zadoff-Chu Sequence Generation

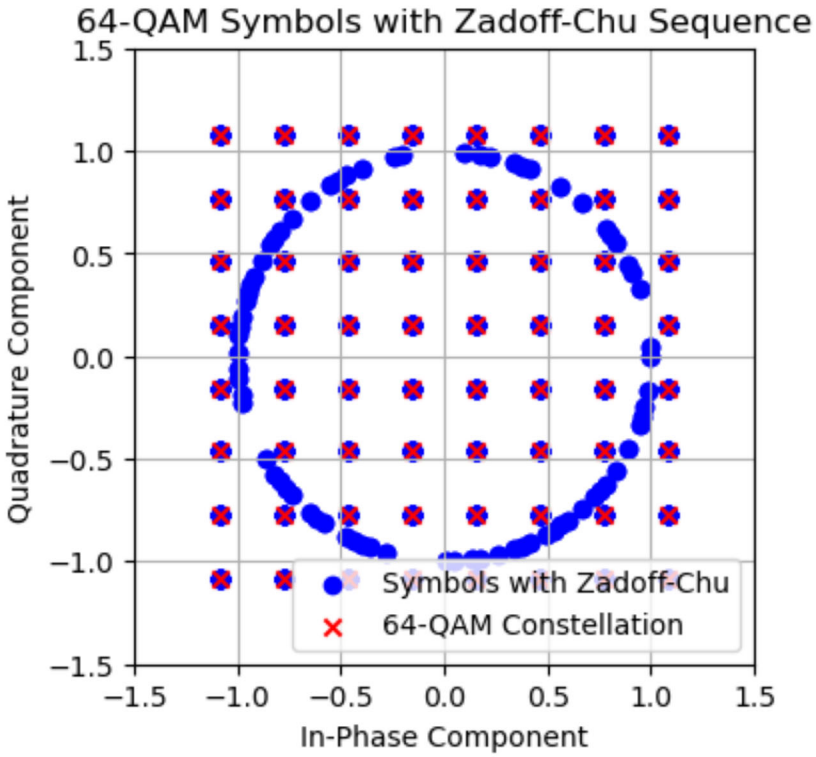
In [10]:
```
# Generate Zadoff-Chu sequence
N_zc = 150 # Length of Zadoff-Chu sequence
N_zc, zc_sequence = gen_zadoff_chu_sequence(N_zc, 23)

# Visualize Zadoff-Chu autocorrelation
plt.figure(figsize=(10, 4))
plt.plot(np.abs(np.correlate(zc_sequence, zc_sequence, mode='full')), label='Zadoff
plt.title("Zadoff-Chu Sequence Autocorrelation")
plt.xlabel("Lag")
plt.ylabel("Autocorrelation Magnitude")
plt.grid(True)
plt.legend()
plt.show()
```

```
Warning: N is not prime, choosing the largest prime less than N: 149.
```



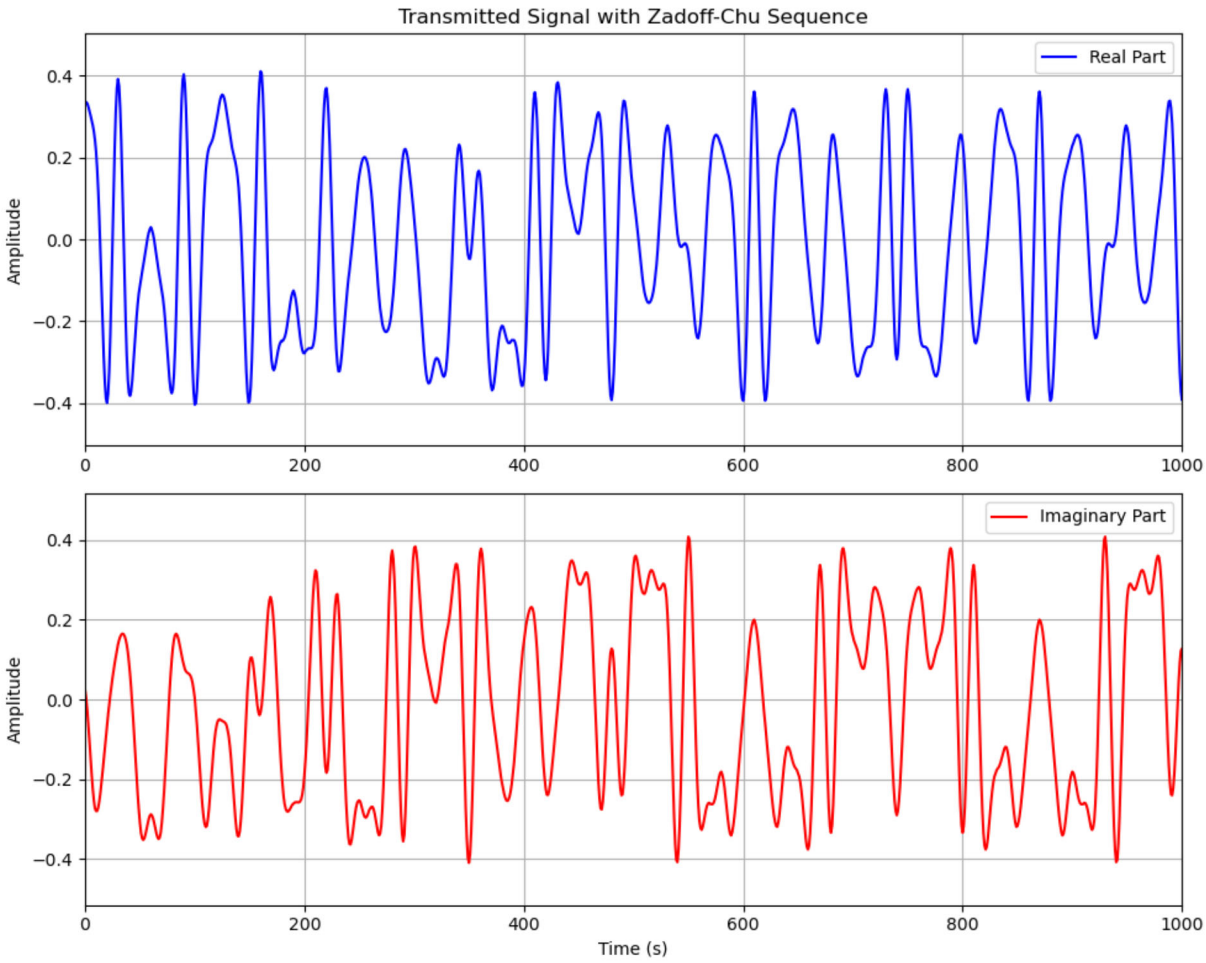
TX Signal Generation

In [11]:
```
# Generate TX signal with Zadoff-Chu sequence
symbols_zc = np.concatenate((zc_sequence, symbols))

# Visualize symbols with Zadoff-Chu sequence
plt.figure(figsize=(4, 4))
plt.scatter(np.real(symbols_zc), np.imag(symbols_zc), c='blue', marker='o', label='
plt.scatter(constellation.real, constellation.imag, c='red', marker='x', label='64-
plt.title("64-QAM Symbols with Zadoff-Chu Sequence")
plt.xlabel("In-Phase Component")
plt.ylabel("Quadrature Component")
plt.grid(True)
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.legend()
plt.show()
```



In [12]:
```
# Pulse shape the Zadoff-Chu sequence
t, pulse_train_zc = create_pulse_train(symbols_zc, sps)
tx_signal_zc = np.convolve(g_tx, pulse_train_zc, mode='same')

# Visualize pulse-shaped TX signal
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Transmitted Signal with Zadoff-Chu Sequence")
plt.plot(t, np.real(tx_signal_zc), label='Real Part', color='blue')
plt.xlim(0, N/10*sps)
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(t, np.imag(tx_signal_zc), label='Imaginary Part', color='red')
plt.xlabel("Time (s)")
```

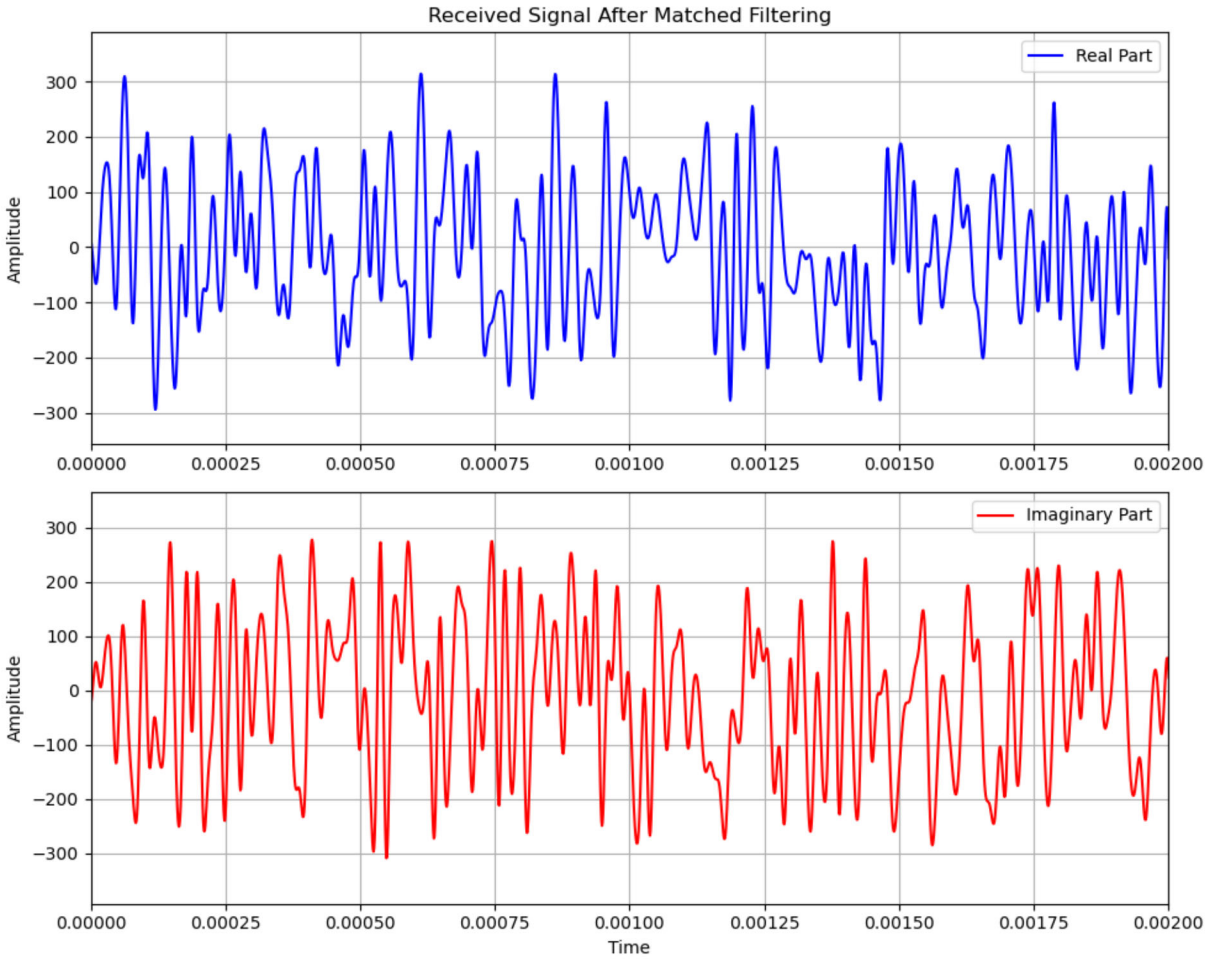
```
plt.ylabel("Amplitude")
plt.xlim(0, N/10*sps)
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```



Signal TX/RX

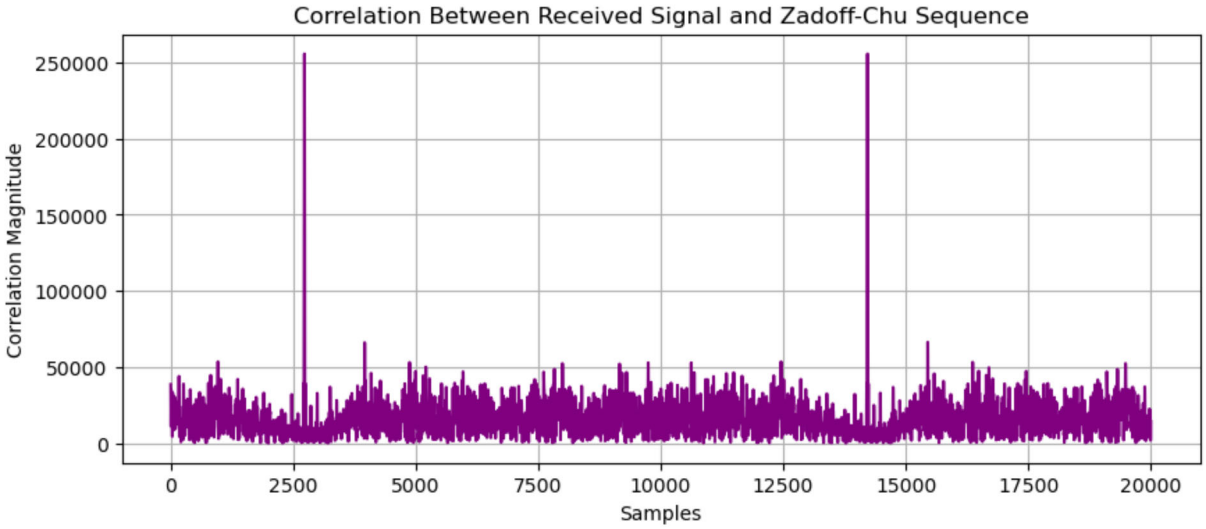
In [13]:
```
# Transmit and receive the signal
if pluto_token is not None:
    # Transmit signal
    tx_scaled_zc = tx_signal_zc / np.max(np.abs(tx_signal_zc)) * 2**14 # Pluto expe
    sdr.tx_destroy_buffer()                    # reset transmit data buffer to be sa
    sdr.tx(tx_scaled_zc)

    # Receive signal
    sdr.rx_destroy_buffer()
    rx_signal_zc = sdr.rx()

    t = np.arange(rx_buffer_size) / sample_rate  # time vector for TX signal

else:
    N0 = 0.001
    zeropad_length = np.random.randint(10, 30)  # Random length for zero padding
    h = np.random.randint(1, 200) * (np.random.randn(1) + 1j * np.random.randn(1))
```

```
    print(f"No Pluto token provided\nSimulating transmission and reception in a fre
    noise = np.sqrt(N0/2) * (np.random.randn(len(tx_signal_zc)) + 1j * np.random.ra
    rx_signal_zc = np.concatenate((np.zeros(zeropad_length*sps), h * tx_signal_zc +
    rx_signal_zc = np.concatenate((rx_signal_zc, rx_signal_zc, rx_signal_zc))
    t = np.arange(len(rx_signal_zc)) / sample_rate  # time vector for TX signal

# Pass received signal through matched filter
rx_signal_mf_zc = np.convolve(rx_signal_zc, g_rx, mode='same')

# Visualize matched filter output
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Received Signal After Matched Filtering")
plt.plot(t, np.real(rx_signal_mf_zc), label='Real Part', color='blue')
plt.ylabel("Amplitude")
plt.xlim(0, t[2*N//10*sps])
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(t, np.imag(rx_signal_mf_zc), label='Imaginary Part', color='red')
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.xlim(0, t[2*N//10*sps])
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```



Correlation with Zadoff-Chu

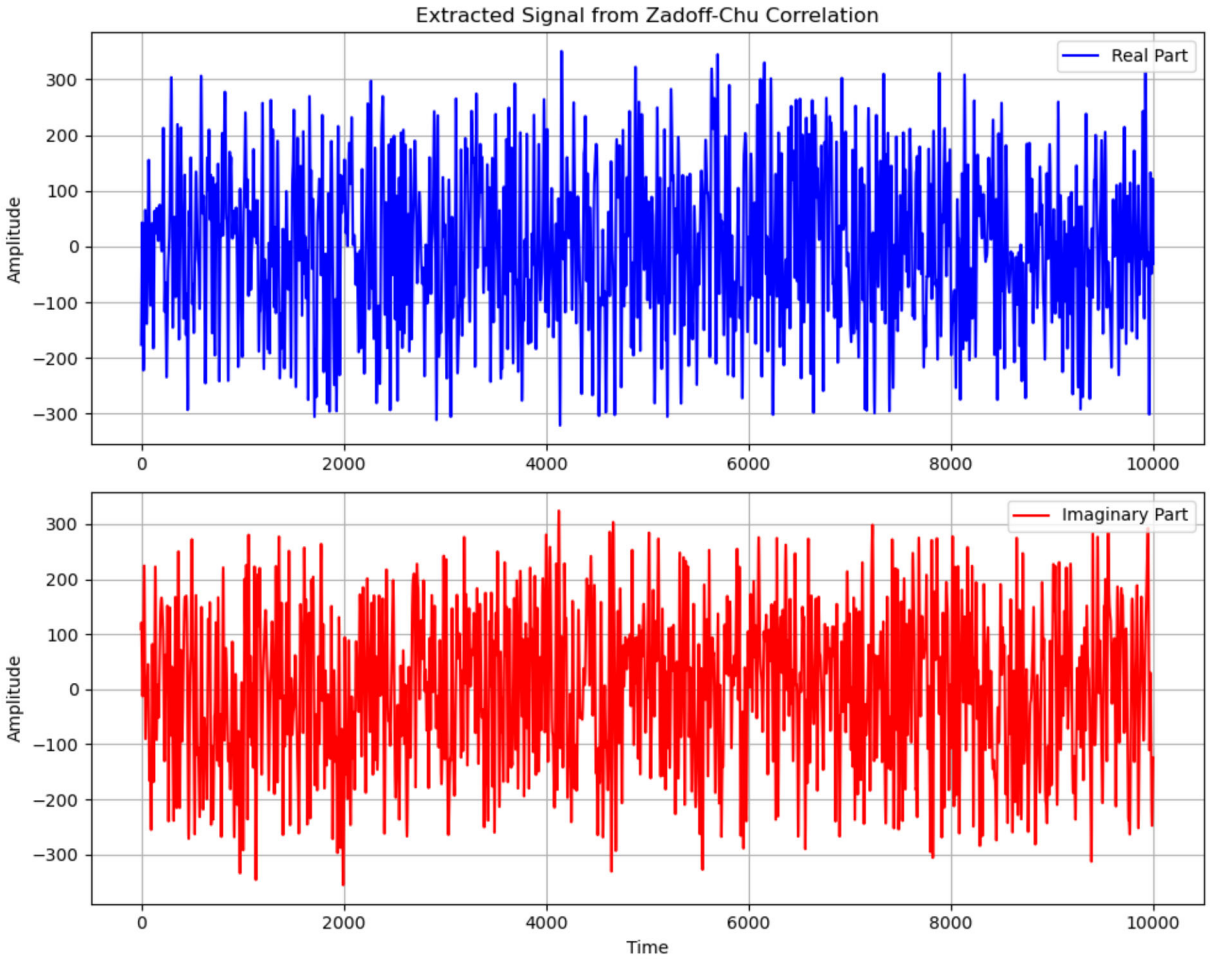
In [14]:
```
# Correlate received signal with pulse-shaped Zadoff-Chu sequence
t_zc_pulse, zc_sequence_pulse = create_pulse_train(zc_sequence, sps)
zc_sequence_shaped = np.convolve(np.convolve(g_tx, zc_sequence_pulse, mode='same'),

correlation = np.correlate(rx_signal_mf_zc, zc_sequence_shaped, mode='valid')

# Visualize correlation result
plt.figure(figsize=(10, 4))
plt.plot(np.abs(correlation[:int(2*N*sps)]), label='Correlation with Zadoff-Chu Seq
plt.title("Correlation Between Received Signal and Zadoff-Chu Sequence")
plt.xlabel("Samples")
plt.ylabel("Correlation Magnitude")
plt.grid(True)
plt.show()
```



Frame Synchronization From Correlation Peaks

In [15]:
```
# Extract one copy of TX signal between correlation peaks
corr_peaks, _ = signal.find_peaks(np.abs(correlation), height=np.max(np.abs(correla
print(f"peak indices: {corr_peaks[0]}, {corr_peaks[1]}")

rx_signal_zc_extracted = rx_signal_mf_zc[N_zc*sps+corr_peaks[0]:corr_peaks[1]]  # E

# Visualize extracted signal
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Extracted Signal from Zadoff-Chu Correlation")
plt.plot(np.real(rx_signal_zc_extracted), label='Real Part', color='blue')
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(np.imag(rx_signal_zc_extracted), label='Imaginary Part', color='red')
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```

```
peak indices: 2728, 14218
```



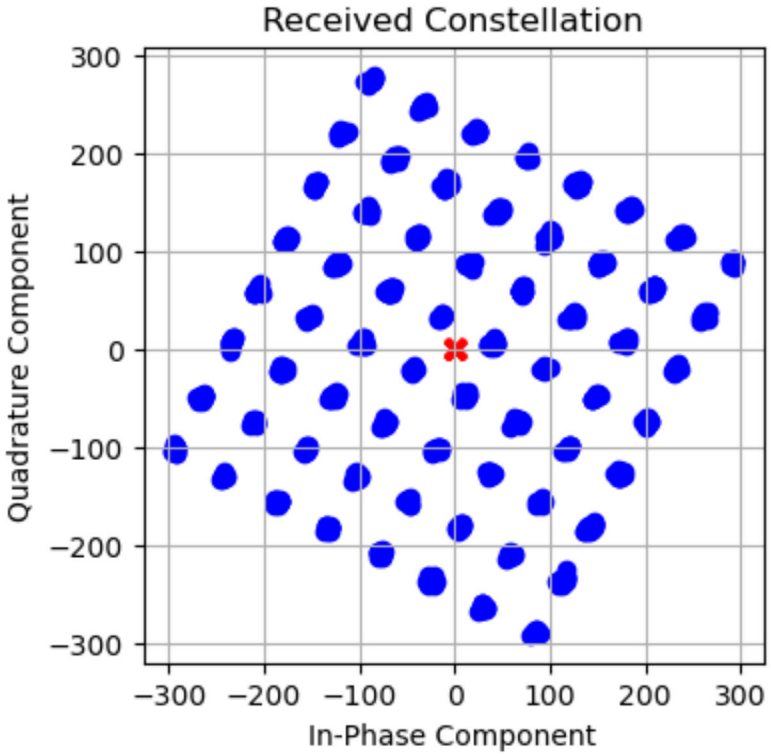
Part V: Channel Equalization and Symbol Detection

Sample the extracted portion of the received signal from Part IV every T seconds to get your received symbols. If everything has gone according to plan, these received symbols contain the transmitted symbols plus AWGN, with little to no ISI. The channel across our narrowband transmission is approximately frequency-flat and thus causes a scaling and rotation of our transmitted symbols. **Plot** the real and imaginary components of each received symbol on the complex plane, and then overlay the originally transmitted symbols. You should be able to observe the effects of the channel.

Received Symbols without Equalization

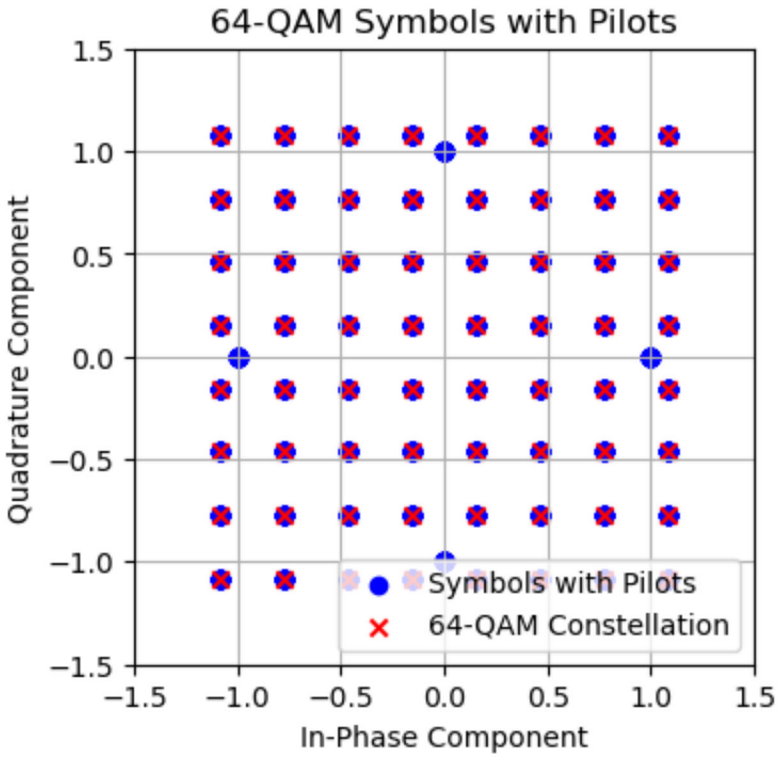
In [16]:
```
# Sample the extracted signal at symbol rate
rx_signal_zc_sampled = rx_signal_zc_extracted[::int(T/ts)]

# Visualize sampled signal
plt.figure(figsize=(4, 4))
plt.scatter(np.real(rx_signal_zc_sampled), np.imag(rx_signal_zc_sampled), c='blue',
plt.scatter(constellation.real, constellation.imag, c='red', marker='x', label=f'{M
plt.title(f"Received Constellation")
plt.xlabel("In-Phase Component")
plt.ylabel("Quadrature Component")
plt.grid(True)
```

```
plt.show()
```



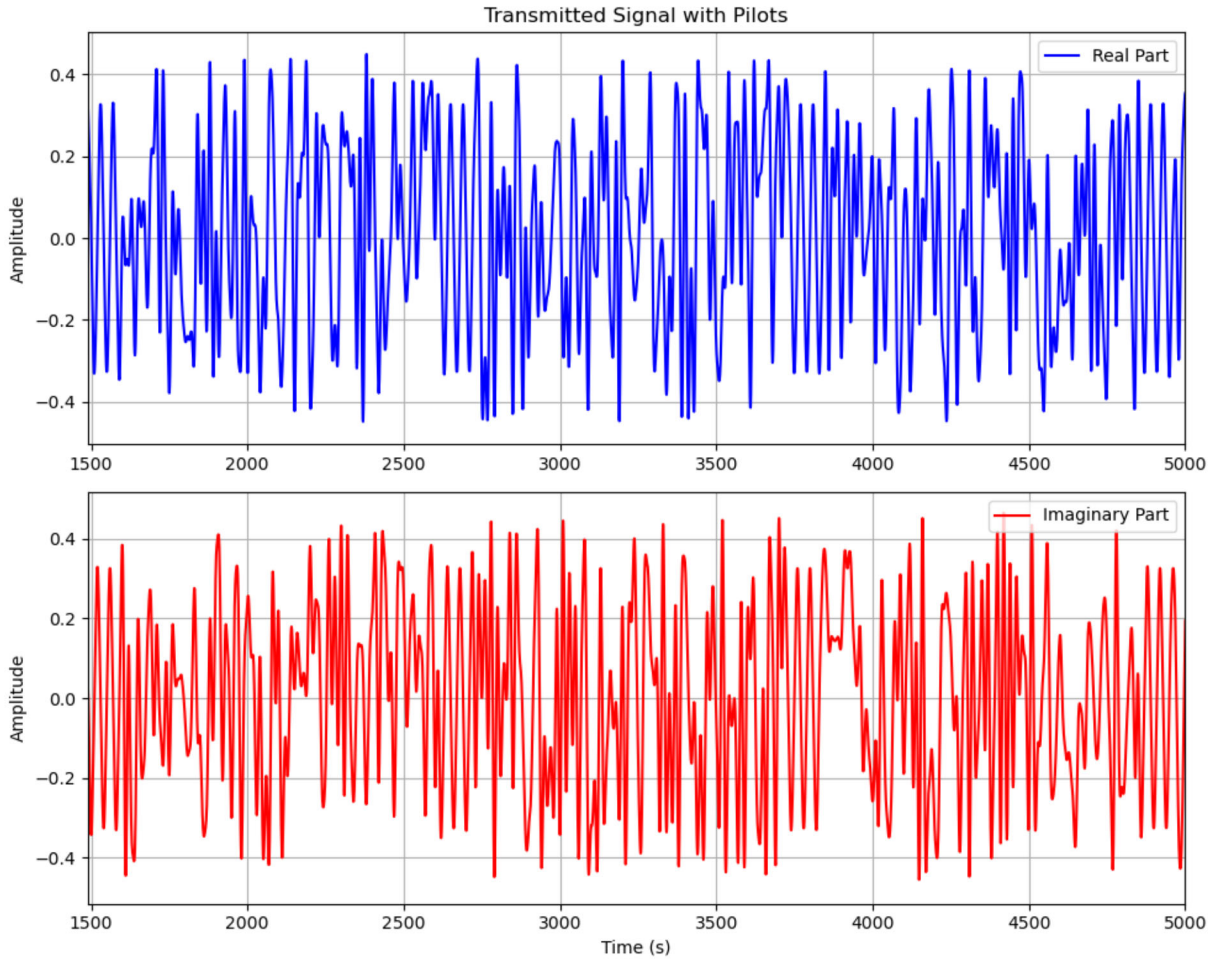
Insert Pilots

In [17]:
```
# Define pilots to be inserted into TX signal
pilot_pattern = np.array([1+0j, 0-1j, -1+0j, 0+1j], dtype=complex) # Pilot symbols
pilot_interval = 100 # Number of symbols between pilots
pilot_n = 12 # Number of pilots to be inserted (after Z-C and before data)
pilot_sequence = np.tile(pilot_pattern, (pilot_n + 3) // 4)[:pilot_n] # Create pilo

# Insert pilots into TX signal
symbols_pilots = []
for i in range(0, len(symbols), pilot_interval):
    symbols_pilots.append(pilot_sequence)
    symbols_pilots.append(symbols[i:i+pilot_interval])
symbols_pilots = np.concatenate(symbols_pilots).astype(complex)

# Visualize symbols with inserted pilots
plt.figure(figsize=(4, 4))
plt.scatter(np.real(symbols_pilots), np.imag(symbols_pilots), c='blue', marker='o',
plt.scatter(constellation.real, constellation.imag, c='red', marker='x', label=f'{M
plt.title(f"{M}-QAM Symbols with Pilots")
plt.xlabel("In-Phase Component")
plt.ylabel("Quadrature Component")
plt.grid(True)
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.legend()
plt.show()
```



TX Signal Generation

In [18]:
```
# Prepend Zadoff-Chu sequence to symbols
symbols_pilots = np.concatenate((zc_sequence, symbols_pilots))

# Pulse shape the new transmit signal
t, pulse_train_pilots = create_pulse_train(symbols_pilots, sps)
tx_signal_pilots = np.convolve(g_tx, pulse_train_pilots, mode='same')

# Visualize pulse-shaped TX signal
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Transmitted Signal with Pilots")
plt.plot(t, np.real(tx_signal_pilots), label='Real Part', color='blue')
plt.xlim(N_zc*sps, N/2*sps)
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(t, np.imag(tx_signal_pilots), label='Imaginary Part', color='red')
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.xlim(N_zc*sps, N/2*sps)
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```

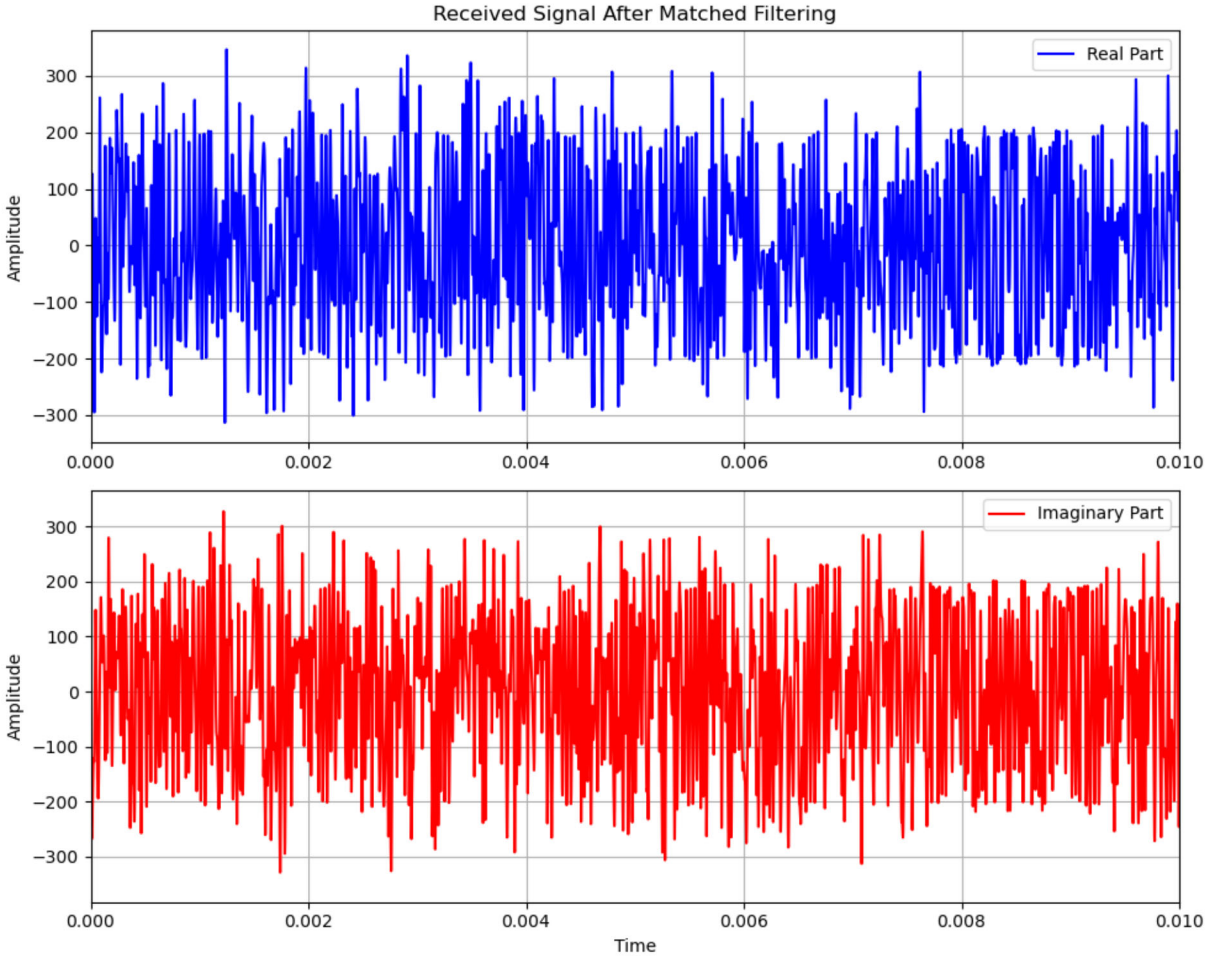


Signal TX/RX

In [19]:
```
# Transmit and receive the signal
if pluto_token is not None:
    print("Transmitting signal on Pluto...")
    # Transmit signal
    tx_scaled_pilots = tx_signal_pilots / np.max(np.abs(tx_signal_pilots)) * 2**14
    sdr.tx_destroy_buffer()                    # reset transmit data buffer to be sa
    sdr.tx(tx_scaled_pilots)

    # Receive signal
    sdr.rx_destroy_buffer()
    rx_signal_pilots = sdr.rx()

    t = np.arange(rx_buffer_size) / sample_rate  # time vector for TX signal

else:
    N0 = 1
    zeropad_length = np.random.randint(10, 30)  # Random length for zero padding
    h = np.random.randint(1, 200) * (np.random.randn(1) + 1j * np.random.randn(1))
    print(f"No Pluto token provided\nSimulating transmission and reception in a fre
    noise = np.sqrt(N0/2) * (np.random.randn(len(tx_signal_pilots)) + 1j * np.rando
    rx_signal_pilots = np.concatenate((np.zeros(zeropad_length*sps), h * tx_signal_
    rx_signal_pilots = np.concatenate((rx_signal_pilots, rx_signal_pilots, rx_signa
    t = np.arange(len(rx_signal_pilots)) / sample_rate  # time vector for TX signal
```

```
# Pass received signal through matched filter
rx_signal_mf_pilots = np.convolve(rx_signal_pilots, g_rx, mode='same')

# Visualize matched filter output
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Received Signal After Matched Filtering")
plt.plot(t, np.real(rx_signal_mf_pilots), label='Real Part', color='blue')
plt.ylabel("Amplitude")
plt.xlim(0, t[N*sps])
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(t, np.imag(rx_signal_mf_pilots), label='Imaginary Part', color='red')
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.xlim(0, t[N*sps])
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```

```
Transmitting signal on Pluto...
```

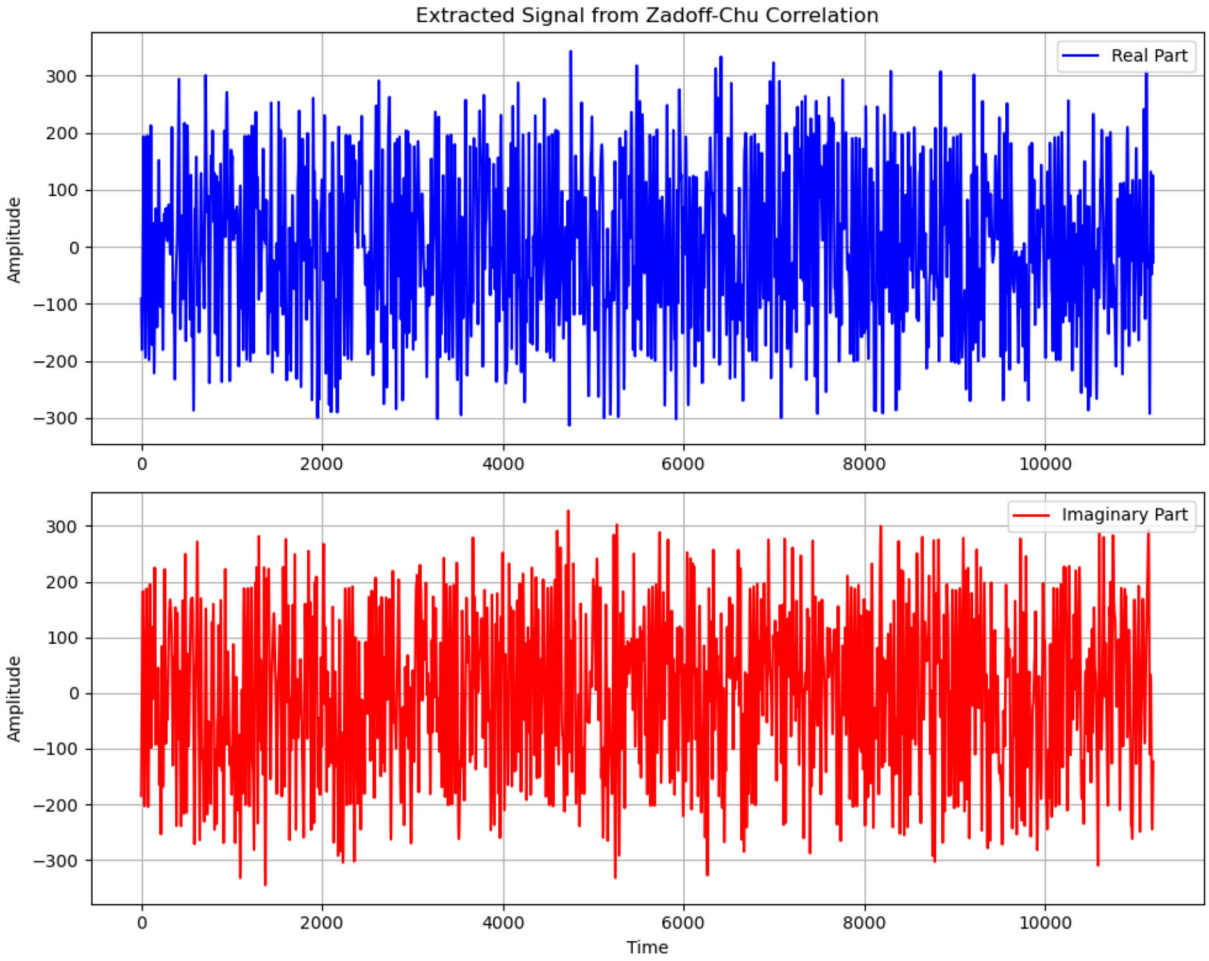


Zadoff-Chu Frame Synchronization

In [20]:
```
# Correlate received signal with pulse-shaped Zadoff-Chu sequence
```

```
_, zc_sequence_pulse = create_pulse_train(zc_sequence, sps)
zc_sequence_shaped = np.convolve(np.convolve(g_tx, zc_sequence_pulse, mode='same'),

correlation = np.correlate(rx_signal_mf_pilots, zc_sequence_shaped, mode='valid')

# Extract one copy of TX signal between correlation peaks
corr_peaks, _ = signal.find_peaks(np.abs(correlation), height=np.max(np.abs(correla
print(f"peak indices: {corr_peaks[0]}, {corr_peaks[1]}")

rx_signal_pilots_extracted = rx_signal_mf_pilots[N_zc*sps+corr_peaks[0]:corr_peaks[

# Visualize extracted signal
plt.figure(figsize=(10, 8))
plt.subplot(2, 1, 1)
plt.title("Extracted Signal from Zadoff-Chu Correlation")
plt.plot(np.real(rx_signal_pilots_extracted), label='Real Part', color='blue')
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend(loc='upper right')
plt.subplot(2, 1, 2)
plt.plot(np.imag(rx_signal_pilots_extracted), label='Imaginary Part', color='red')
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend(loc='upper right')
plt.tight_layout()
plt.show()
```

```
peak indices: 7695, 20385
```

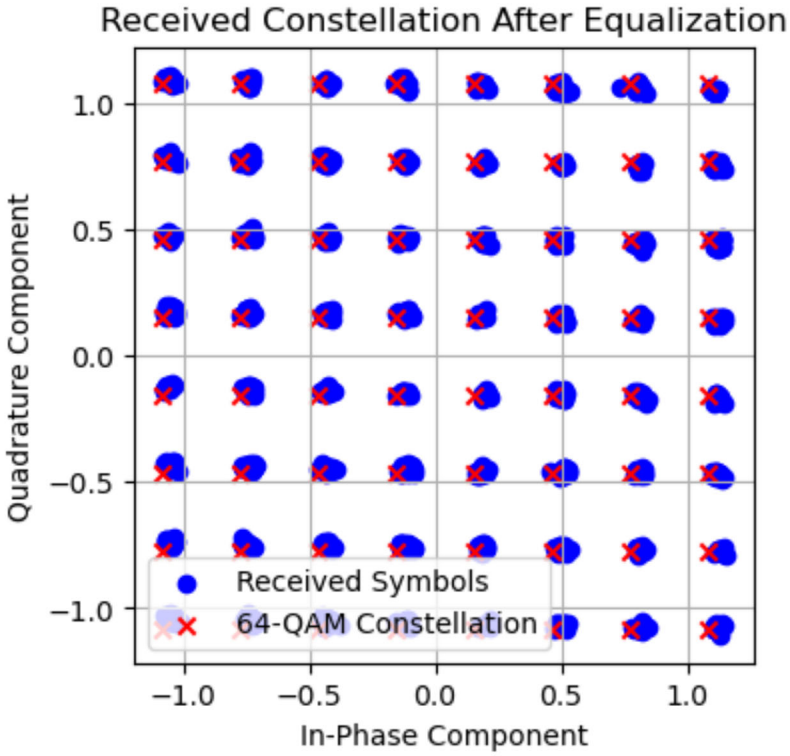


Channel Estimation and Equalization

In [21]:
```
# Sample extracted signal at symbol rate
rx_signal_pilots_sampled = rx_signal_pilots_extracted[::sps]

# Estimate and equalize channel using pilots
for i in range(int(N/pilot_interval)):
    start = i * (pilot_n + pilot_interval)
    end = start + pilot_n
    h_est = np.mean(rx_signal_pilots_sampled[start:end] / pilot_sequence)  # Estima
    rx_signal_pilots_sampled[start:end+pilot_interval] /= h_est  # Equalize channel
    print(f"Estimated channel gain for pilot {i}: {h_est}")

# Remove pilots from sampled signal
rx_signal_pilots_sampled_trimmed = []
for i in range(int(N/pilot_interval)):
    start = i * (pilot_n + pilot_interval)
    rx_signal_pilots_sampled_trimmed.append(rx_signal_pilots_sampled[start + pilot_
rx_signal_pilots_sampled_trimmed = np.concatenate(rx_signal_pilots_sampled_trimmed)

# Visualize sampled signal
plt.figure(figsize=(4, 4))
plt.scatter(np.real(rx_signal_pilots_sampled_trimmed), np.imag(rx_signal_pilots_sam
plt.scatter(constellation.real, constellation.imag, c='red', marker='x', label=f'{M
plt.title(f"Received Constellation After Equalization")
plt.xlabel("In-Phase Component")
```

```
plt.ylabel("Quadrature Component")
plt.grid(True)
plt.legend()
plt.show()
```

```
Estimated channel gain for pilot 0: (-88.22233639314341-174.61478565785654j)
Estimated channel gain for pilot 1: (-88.62719702577115-175.11590640523684j)
Estimated channel gain for pilot 2: (-87.95130015689972-175.1515301546205j)
Estimated channel gain for pilot 3: (-88.13083669735579-175.06299931541736j)
Estimated channel gain for pilot 4: (-88.9573367342153-175.0119272736565j)
Estimated channel gain for pilot 5: (-88.06818668405836-175.4510449284548j)
Estimated channel gain for pilot 6: (-88.4238896958571-175.33435460816477j)
Estimated channel gain for pilot 7: (-88.63488816240971-174.64620868569597j)
Estimated channel gain for pilot 8: (-88.67376716782385-175.3154865588073j)
Estimated channel gain for pilot 9: (-88.6499021807779-174.45077018024023j)
```



Symbol Detection and SER Calculation

In [22]:
```
# Perform maximum likelihood symbol detection
rx_symbols_detected = []
for symbol in rx_signal_pilots_sampled_trimmed:
    distances = np.abs(symbol - constellation)  # Calculate distances to constellat
    closest_symbol = constellation[np.argmin(distances)]  # Find closest constellat
    rx_symbols_detected.append(closest_symbol)
rx_symbols_detected = np.array(rx_symbols_detected)

# Calculate symbol error rate
ser = np.mean(symbols != rx_symbols_detected[:len(symbols)])  # Symbol error rate
print(f"Symbol Error Rate (SER): {ser:.4f}")
```

```
Symbol Error Rate (SER): 0.0000
```

Part VI: Symbol Error Rate Curves

Vary the transmit gain from `-50 dB` to `-25 dB` and **plot** the corresponding SER for each. Do not exceed `-25 dB`. **Repeat** this for 256-QAM and include it on the same plot as 64-QAM. The y-axis should be in log-scale and the x-axis should be transmit gain in dB. Note that, in order to get meaningful values of SER, you will likely need to average across multiple transmissions.

Full System Function

In [60]:
```
def albert (M, tx_gain, sdr):
    """
    This function simulates the full digital communication system designed in parts
    1. Random M-QAM symbol generation
    2. Zadoff-Chu sequence generation
    3. Pilot sequence insertion
    4. Pulse shaping using a root raised cosine filter
    5. Transmission and reception using Pluto SDR
    6. Matched filtering of the received signal
    7. Frame synchronization using Zadoff-Chu sequence correlation
    8. Channel estimation and equalization using pilot symbols
    9. Maximum Likelihood symbol detection
    10. Symbol error rate calculation
    This function returns the symbol error rate.
    """
    # System parameters
    fs = 1e6     # baseband sampling rate (samples per second)
    ts = 1 / fs  # baseband sampling period (seconds)
    sps = 10     # samples per data symbol
    T = ts * sps # time between data symbols (seconds per symbol)

    # SDR parameters
    if (tx_gain > -25):
        raise ValueError("tx_gain must be less than or equal to -25 dB.")
    if (sdr is None):
        raise ValueError("sdr object must be provided to use Pluto SDR.")
    sample_rate = fs                # sampling rate, between ~600e3 and 61e6
    tx_carrier_freq_Hz = 915e6      # transmit carrier frequency, between 325 MHz t
    rx_carrier_freq_Hz = 915e6      # receive carrier frequency, between 325 MHz to
    tx_rf_bw_Hz = sample_rate * 1   # transmitter's RF bandwidth, between 200 kHz a
    rx_rf_bw_Hz = sample_rate * 1   # receiver's RF bandwidth, between 200 kHz and
    tx_gain_dB = tx_gain            # transmit gain (in dB), between -89.75 to 0 dB
    rx_gain_dB = 40                 # receive gain (in dB), between 0 to 74.5 dB (o
    rx_agc_mode = 'manual'          # receiver's AGC mode: 'manual', 'slow_attack',
    rx_buffer_size = 500e3          # receiver's buffer size (in samples), length o
    tx_cyclic_buffer = True         # cyclic nature of transmitter's buffer (True -
    if sdr is not None:
        sdr.sample_rate = int(sample_rate)   # set baseband sampling rate of Pluto
        sdr.tx_destroy_buffer()                    # reset transmit data buffer to b
        sdr.tx_rf_bandwidth = int(tx_rf_bw_Hz)     # set transmitter RF bandwidth
        sdr.tx_lo = int(tx_carrier_freq_Hz)        # set carrier frequency for trans
        sdr.tx_hardwaregain_chan0 = tx_gain_dB     # set the transmit gain
        sdr.tx_cyclic_buffer = tx_cyclic_buffer    # set the cyclic nature of the tr
        sdr.rx_destroy_buffer()                    # reset receive data buffer to be
        sdr.rx_lo = int(rx_carrier_freq_Hz)        # set carrier frequency for recep
```

```
    sdr.rx_rf_bandwidth = int(sample_rate)    # set receiver RF bandwidth
    sdr.rx_buffer_size = int(rx_buffer_size)  # set buffer size of receiver
    sdr.gain_control_mode_chan0 = rx_agc_mode # set gain control mode
    sdr.rx_hardwaregain_chan0 = rx_gain_dB    # set gain of receiver

# Symbol generation
N=1000
symbols, constellation = gen_rand_qam_symbols(N, M)

# Zadoff-Chu sequence generation
N_zc = 149
N_zc, zc_sequence = gen_zadoff_chu_sequence(N_zc, 23)

# Pilot sequence generation
pilot_pattern = np.array([1+0j, 0-1j, -1+0j, 0+1j], dtype=complex)
pilot_interval = 100
pilot_n = 12
pilot_sequence = np.tile(pilot_pattern, (pilot_n + 3) // 4)[:pilot_n]

# TX signal generation
_, g_tx = get_rrc_pulse(0.5, 8, 10)  # Root raised cosine pulse
tx_symbols = []
for i in range(0, len(symbols), pilot_interval):
    tx_symbols.append(pilot_sequence)
    tx_symbols.append(symbols[i:i+pilot_interval])
tx_symbols = np.concatenate(tx_symbols).astype(complex)
tx_symbols = np.concatenate((zc_sequence, tx_symbols))
_, pulse_train = create_pulse_train(tx_symbols, 10)
tx_signal = np.convolve(g_tx, pulse_train, mode='same')

# Transmit/receive via Pluto SDR
tx_scaled = tx_signal / np.max(np.abs(tx_signal)) * 2**14  # Scale for Pluto
sdr.tx_destroy_buffer()
sdr.rx_destroy_buffer()
sdr.tx(tx_scaled)
rx_signal = sdr.rx()

# Matched filtering
g_rx = np.conj(g_tx[::-1])  # Matched filter
rx_signal_mf = np.convolve(rx_signal, g_rx, mode='same')

# Frame synchronization
_, zc_sequence_pulse = create_pulse_train(zc_sequence, sps)
zc_sequence_shaped = np.convolve(np.convolve(g_tx, zc_sequence_pulse, mode='sam
correlation = np.correlate(rx_signal_mf, zc_sequence_shaped, mode='valid')
corr_peaks, _ = signal.find_peaks(np.abs(correlation), height=np.max(np.abs(cor
rx_signal_extracted = rx_signal_mf[N_zc*sps+corr_peaks[0]:corr_peaks[1]]

# Sample signal
rx_signal_sampled = rx_signal_extracted[::sps]

# Estimate and equalize channel using pilots
for i in range(int(N/pilot_interval)):
    start = i * (pilot_n + pilot_interval)
    end = start + pilot_n
    h_est = np.mean(rx_signal_sampled[start:end] / pilot_sequence)  # Estimate
```

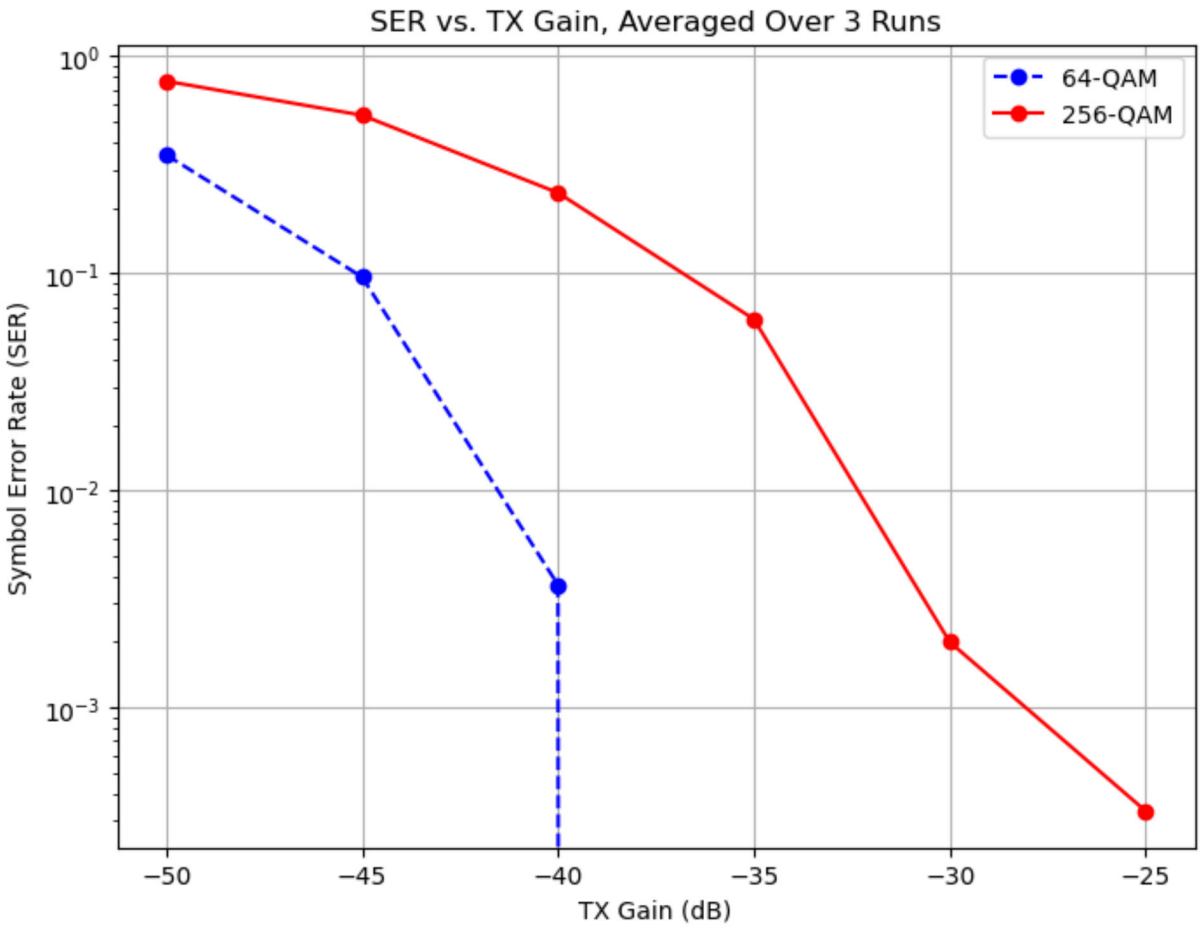
```
        rx_signal_sampled[start:end+pilot_interval] /= h_est  # Equalize channel

    # Remove pilots from sampled signal
    rx_signal_trimmed = []
    for i in range(int(N/pilot_interval)):
        start = i * (pilot_n + pilot_interval)
        rx_signal_trimmed.append(rx_signal_sampled[start + pilot_n:start + pilot_n
    rx_signal_trimmed = np.concatenate(rx_signal_trimmed)

    # Perform maximum likelihood symbol detection
    rx_symbols = []
    for symbol in rx_signal_trimmed:
        distances = np.abs(symbol - constellation)  # Calculate distances to conste
        closest_symbol = constellation[np.argmin(distances)]  # Find closest conste
        rx_symbols.append(closest_symbol)
    rx_symbols = np.array(rx_symbols)

    # Calculate symbol error rate
    ser = np.mean(symbols != rx_symbols[:len(symbols)])  # Symbol error rate

    print(f"finished {M}-QAM, {tx_gain} dB, SER: {ser:.4f}")

    return ser
```

SER vs. TX gain

In [67]:
```
# Sweep TX gain and measure SER

ser_64 = []
ser_256 = []
if pluto_token is not None:
    sdr = adi.Pluto(token=pluto_token) # create Pluto object

step = 5
for i in range(-50, -20, step):
    ser64 = 0
    ser256 = 0
    n_reps = 3
    for j in range(n_reps):
        ser64 += albert(64, i, sdr)
        ser256 += albert(256, i, sdr)
    ser_64.append(ser64 / n_reps)
    ser_256.append(ser256 / n_reps)
```

```
finished 64-QAM, -50 dB, SER: 0.4270
finished 256-QAM, -50 dB, SER: 0.8060
finished 64-QAM, -50 dB, SER: 0.3080
finished 256-QAM, -50 dB, SER: 0.7540
finished 64-QAM, -50 dB, SER: 0.3160
finished 256-QAM, -50 dB, SER: 0.7360
finished 64-QAM, -45 dB, SER: 0.1200
finished 256-QAM, -45 dB, SER: 0.5690
finished 64-QAM, -45 dB, SER: 0.1190
finished 256-QAM, -45 dB, SER: 0.5600
finished 64-QAM, -45 dB, SER: 0.0490
finished 256-QAM, -45 dB, SER: 0.4710
finished 64-QAM, -40 dB, SER: 0.0020
finished 256-QAM, -40 dB, SER: 0.2600
finished 64-QAM, -40 dB, SER: 0.0050
finished 256-QAM, -40 dB, SER: 0.2390
finished 64-QAM, -40 dB, SER: 0.0040
finished 256-QAM, -40 dB, SER: 0.2040
finished 64-QAM, -35 dB, SER: 0.0000
finished 256-QAM, -35 dB, SER: 0.0000
finished 64-QAM, -35 dB, SER: 0.0000
finished 256-QAM, -35 dB, SER: 0.0990
finished 64-QAM, -35 dB, SER: 0.0000
finished 256-QAM, -35 dB, SER: 0.0850
finished 64-QAM, -30 dB, SER: 0.0000
finished 256-QAM, -30 dB, SER: 0.0000
finished 64-QAM, -30 dB, SER: 0.0000
finished 256-QAM, -30 dB, SER: 0.0030
finished 64-QAM, -30 dB, SER: 0.0000
finished 256-QAM, -30 dB, SER: 0.0030
finished 64-QAM, -25 dB, SER: 0.0000
finished 256-QAM, -25 dB, SER: 0.0000
finished 64-QAM, -25 dB, SER: 0.0000
finished 256-QAM, -25 dB, SER: 0.0010
finished 64-QAM, -25 dB, SER: 0.0000
finished 256-QAM, -25 dB, SER: 0.0000
```

In [68]:
```
# Visualize SER vs TX Gain

plt.figure(figsize=(8, 6))
plt.plot(range(-50, -20, step), ser_64, marker='o', color='blue', linestyle='--', l
plt.plot(range(-50, -20, step), ser_256, marker='o', color='red', label='256-QAM')
plt.yscale('log')
plt.title("SER vs. TX Gain, Averaged Over 3 Runs")
plt.xlabel("TX Gain (dB)")
plt.ylabel("Symbol Error Rate (SER)")
plt.grid(True)
plt.legend()
plt.show()
```



Part VII: What Does a Matched Filter Buy Us?

Finally, suppose instead of a root raised cosine filter, we use a raised cosine filter at the transmitter, without a matched filter at the receiver. We still perform all of the aforementioned synchronization and equalization procedures. **Repeat** Part VI, overlaying your lines atop those from Part VI on a single figure. Does the matched filter seem to improve SER? By how much? Pretty cool huh?

Modified System Function Using RC

In [77]:
```
def albert1 (M, tx_gain, sdr):
    """
    This function simulates the full digital communication system designed in parts
    This function returns the symbol error rate.
    """
    # System parameters
    fs = 1e6     # baseband sampling rate (samples per second)
    ts = 1 / fs  # baseband sampling period (seconds)
    sps = 10     # samples per data symbol
    T = ts * sps # time between data symbols (seconds per symbol)

    # SDR parameters
    if (tx_gain > -25):
        raise ValueError("tx_gain must be less than or equal to -25 dB.")
```

```
if (sdr is None):
    raise ValueError("sdr object must be provided to use Pluto SDR.")
sample_rate = fs                # sampling rate, between ~600e3 and 61e6
tx_carrier_freq_Hz = 915e6      # transmit carrier frequency, between 325 MHz t
rx_carrier_freq_Hz = 915e6      # receive carrier frequency, between 325 MHz to
tx_rf_bw_Hz = sample_rate * 1   # transmitter's RF bandwidth, between 200 kHz a
rx_rf_bw_Hz = sample_rate * 1   # receiver's RF bandwidth, between 200 kHz and
tx_gain_dB = tx_gain            # transmit gain (in dB), between -89.75 to 0 dB
rx_gain_dB = 40                 # receive gain (in dB), between 0 to 74.5 dB (o
rx_agc_mode = 'manual'          # receiver's AGC mode: 'manual', 'slow_attack',
rx_buffer_size = 500e3          # receiver's buffer size (in samples), length o
tx_cyclic_buffer = True         # cyclic nature of transmitter's buffer (True -
if sdr is not None:
    sdr.sample_rate = int(sample_rate)   # set baseband sampling rate of Pluto
    sdr.tx_destroy_buffer()                    # reset transmit data buffer to b
    sdr.tx_rf_bandwidth = int(tx_rf_bw_Hz)     # set transmitter RF bandwidth
    sdr.tx_lo = int(tx_carrier_freq_Hz)        # set carrier frequency for trans
    sdr.tx_hardwaregain_chan0 = tx_gain_dB     # set the transmit gain
    sdr.tx_cyclic_buffer = tx_cyclic_buffer    # set the cyclic nature of the tr
    sdr.rx_destroy_buffer()                    # reset receive data buffer to be
    sdr.rx_lo = int(rx_carrier_freq_Hz)        # set carrier frequency for recep
    sdr.rx_rf_bandwidth = int(sample_rate)     # set receiver RF bandwidth
    sdr.rx_buffer_size = int(rx_buffer_size)   # set buffer size of receiver
    sdr.gain_control_mode_chan0 = rx_agc_mode # set gain control mode
    sdr.rx_hardwaregain_chan0 = rx_gain_dB     # set gain of receiver

# Symbol generation
N=1000
symbols, constellation = gen_rand_qam_symbols(N, M)

# Zadoff-Chu sequence generation
N_zc = 149
N_zc, zc_sequence = gen_zadoff_chu_sequence(N_zc, 23)

# Pilot sequence generation
pilot_pattern = np.array([1+0j, 0-1j, -1+0j, 0+1j], dtype=complex)
pilot_interval = 100
pilot_n = 12
pilot_sequence = np.tile(pilot_pattern, (pilot_n + 3) // 4)[:pilot_n]

# TX signal generation
_, g_tx = get_rc_pulse(0.5, 8, 10)  # Raised cosine pulse
tx_symbols = []
for i in range(0, len(symbols), pilot_interval):
    tx_symbols.append(pilot_sequence)
    tx_symbols.append(symbols[i:i+pilot_interval])
tx_symbols = np.concatenate(tx_symbols).astype(complex)
tx_symbols = np.concatenate((zc_sequence, tx_symbols))
_, pulse_train = create_pulse_train(tx_symbols, 10)
tx_signal = np.convolve(g_tx, pulse_train, mode='same')

# Transmit/receive via Pluto SDR
sdr.tx_destroy_buffer()
sdr.rx_destroy_buffer()
tx_scaled = tx_signal / np.max(np.abs(tx_signal)) * 2**14  # Scale for Pluto
sdr.tx(tx_scaled)
```

```
    rx_signal = sdr.rx()

    # Matched filtering
    # g_rx = np.conj(g_tx[::-1])  # Matched filter
    # rx_signal_mf = np.convolve(rx_signal, g_rx, mode='same')

    # Frame synchronization
    _, zc_sequence_pulse = create_pulse_train(zc_sequence, sps)
    zc_sequence_shaped = np.convolve(np.convolve(g_tx, zc_sequence_pulse, mode='sam
    correlation = np.correlate(rx_signal, zc_sequence_shaped, mode='valid')
    corr_peaks, _ = signal.find_peaks(np.abs(correlation), height=np.max(np.abs(cor
    rx_signal_extracted = rx_signal[N_zc*sps+corr_peaks[0]:corr_peaks[1]]

    # Sample signal
    rx_signal_sampled = rx_signal_extracted[::sps]

    # Estimate and equalize channel using pilots
    for i in range(int(N/pilot_interval)):
        start = i * (pilot_n + pilot_interval)
        end = start + pilot_n
        h_est = np.mean(rx_signal_sampled[start:end] / pilot_sequence)  # Estimate
        rx_signal_sampled[start:end+pilot_interval] /= h_est  # Equalize channel

    # Remove pilots from sampled signal
    rx_signal_trimmed = []
    for i in range(int(N/pilot_interval)):
        start = i * (pilot_n + pilot_interval)
        rx_signal_trimmed.append(rx_signal_sampled[start + pilot_n:start + pilot_n
    rx_signal_trimmed = np.concatenate(rx_signal_trimmed)

    # Perform maximum likelihood symbol detection
    rx_symbols = []
    for symbol in rx_signal_trimmed:
        distances = np.abs(symbol - constellation)  # Calculate distances to conste
        closest_symbol = constellation[np.argmin(distances)]  # Find closest conste
        rx_symbols.append(closest_symbol)
    rx_symbols = np.array(rx_symbols)

    # Calculate symbol error rate
    ser = np.mean(symbols != rx_symbols[:len(symbols)])  # Symbol error rate

    print(f"finished {M}-QAM, {tx_gain} dB, SER: {ser:.4f}")

    return ser
```

SER vs. TX Gain and MF

In [79]:
```
# Sweep TX gain without MF and measure SER

ser_64_rc = []
ser_256_rc = []
if pluto_token is not None:
    sdr = adi.Pluto(token=pluto_token) # create Pluto object

step = 5
```

```
for i in range(-50, -20, step):
    ser64 = 0
    ser256 = 0
    n_reps = 3
    for j in range(n_reps):
        ser64 += albert1(64, i, sdr)
        ser256 += albert1(256, i, sdr)
    ser_64_rc.append(ser64 / n_reps)
    ser_256_rc.append(ser256 / n_reps)
```

```
finished 64-QAM, -50 dB, SER: 0.6100
finished 256-QAM, -50 dB, SER: 0.8920
finished 64-QAM, -50 dB, SER: 0.6290
finished 256-QAM, -50 dB, SER: 0.9220
finished 64-QAM, -50 dB, SER: 0.6400
finished 256-QAM, -50 dB, SER: 0.8870
finished 64-QAM, -45 dB, SER: 0.3730
finished 256-QAM, -45 dB, SER: 0.7710
finished 64-QAM, -45 dB, SER: 0.3520
finished 256-QAM, -45 dB, SER: 0.7020
finished 64-QAM, -45 dB, SER: 0.3530
finished 256-QAM, -45 dB, SER: 0.6830
finished 64-QAM, -40 dB, SER: 0.0210
finished 256-QAM, -40 dB, SER: 0.3580
finished 64-QAM, -40 dB, SER: 0.0090
finished 256-QAM, -40 dB, SER: 0.4780
finished 64-QAM, -40 dB, SER: 0.0170
finished 256-QAM, -40 dB, SER: 0.3010
finished 64-QAM, -35 dB, SER: 0.0000
finished 256-QAM, -35 dB, SER: 0.0590
finished 64-QAM, -35 dB, SER: 0.0000
finished 256-QAM, -35 dB, SER: 0.1980
finished 64-QAM, -35 dB, SER: 0.0010
finished 256-QAM, -35 dB, SER: 0.1330
finished 64-QAM, -30 dB, SER: 0.0000
finished 256-QAM, -30 dB, SER: 0.1260
finished 64-QAM, -30 dB, SER: 0.0000
finished 256-QAM, -30 dB, SER: 0.0510
finished 64-QAM, -30 dB, SER: 0.0000
finished 256-QAM, -30 dB, SER: 0.0160
finished 64-QAM, -25 dB, SER: 0.0000
finished 256-QAM, -25 dB, SER: 0.0000
finished 64-QAM, -25 dB, SER: 0.0000
finished 256-QAM, -25 dB, SER: 0.0650
finished 64-QAM, -25 dB, SER: 0.0000
finished 256-QAM, -25 dB, SER: 0.0010
```

In [80]:
```
# Visualize SER vs TX Gain

plt.figure(figsize=(8, 6))
plt.plot(range(-50, -20, step), ser_64, marker='o', color='blue', label='64-QAM (MF
plt.plot(range(-50, -20, step), ser_256, marker='o', color='red', label='256-QAM (M
plt.plot(range(-50, -20, step), ser_64_rc, marker='x', color='blue', linestyle='--'
plt.plot(range(-50, -20, step), ser_256_rc, marker='x', color='red', linestyle='--'
plt.yscale('log')
plt.title("SER vs. TX Gain and MF, Averaged Over 3 Runs")
```

```
plt.xlabel("TX Gain (dB)")
plt.ylabel("Symbol Error Rate (SER)")
plt.grid(True)
plt.legend()
plt.show()
```