CS 378 Assignment 2
Design Document

Saksham, Geet, Kavya, Dion

Spring 2024

1 Preamble

In order to maintain the reliability and accuracy of signal transmission, we use a preamble. A preamble is a pre-defined sequence of bits transmitted at the beginning of the signal.

In our assignment, the preamble is generated as a sine wave at a distinct frequency, different from those used for transmitting the actual data. This frequency difference allows the receiver to easily identify the preamble and differentiate it from the main message. The preamble is transmitted 10 times, with each one having a short bit duration of typically around 0.02s. This extended transmission time helps the receiver lock onto the signal and synchronize accurately, ensuring that the actual data that follows is received without errors.

2 Synchronization

During signal transmission, synchronization between the sender and the receiver is of utmost importance to make sure that the message it correctly interpreted. Proper synchronization ensures that the receiver correctly aligns its listening window with the sender's bit transmission. Without synchronization, even small timing discrepancies can lead to significant errors in interpreting the transmitted data. To put this into perspective, consider the case where the sender is transmitting the sequence `101001`, where each bit duration is 0.3 seconds. If the receiver starts listening at the wrong time—let's say, 0.4 seconds late—it can miss the first bit, and may not correctly identify the upcoming bits. In real situations, errors due to improper synchronization are quite common. So we address this as follows:

For this, we employ a preamble and design the receiver to detect changes in the transmitted bits. The preamble has a bit duration of approximately 0.02 seconds, while the actual message bits transmitted by the sender have a much longer duration of around 0.3 seconds. The short duration of the preamble relative to the bit duration allows for a degree of tolerance—any slight delay or variation in detecting the preamble will not significantly impact the timing of the subsequent bit transmission.

Once the receiver detects and identifies the preamble, it becomes synchronized with the sender and is prepared to receive the actual data. The longer bit duration ensures that even if there are minor discrepancies in timing, the receiver can still correctly interpret the data.

3 Error Correction

We use a **CRC based** algorithm to check for errors using redundant bits. Our CRC generator polynomial is **0x5d7**, chosen from this online compendium on CRCs (We chose the minimum HD to be 5 to be able to detect upto 2 errors)

3.1 Encoding algorithm

As is standard in CRC codes, we use a `crc_remainder` function that finds the remainder of the input bitstring (appended with (length of generator polynomial - 1) zeroes to the right of it) with the generator polynomial
This `crc_remainder` function does normal long division over the Galois Field $GF(2)$
The pseudocode for the `crc_remainder` function is as follows -

```
crc_remainder(bit-string input, bit-string generator_polynomial):
    let L <- length(generator_polynomial)
    let augmented_code <- input appended with (L - 1) zeros to the right
    for bit i in augmented_code:
        if bit i is 1 and there are atleast (L-1) bits to the right of i:
            XOR the next L bits (including bit i) of augmented_code with generator_polynomial
    return augmented_code
```

Our final encoding algorithm is as follows -

```
encode_crc(bit-string input, bit-string generator_polynomial):
    let rem <- crc_remainder(input, generator_polynomial)
    return (input appended to rem)
```

The fact that our polynomial has degree 11 and our message length can go upto 20 means that our transmitted message can go upto 31 bits in length
Along with the 6 bits in the preamble, the total message length goes up to 37 bits

3.2 Decoding and Error Correcting Algorithm

Our decoding algorithm is a 'trial-and-error' algorithm that works as follows (and uses the fact that there can be atmost 2 errors in the transmitted message):

```
decode(bit-string transmitted_message, bit-string generator_polynomial):
    // Assume no error for now
    if(crc_remainder(transmitted_message, generator_polynomial) == 0):
        return transmitted_message // No error
    // Assuming 1 error now
    for(bit i in transmitted_message):
        augmented_message <- transmitted_message with bit i flipped
        if(crc_remainder(augmented_message, generator_polynomial) == 0):
            return transmitted_message // 1 error
    // Now it has to be 2 errors
    for(bit i and j in transmitted_message):
        augmented_message <- transmitted_message with bits i and j flipped
        if(crc_remainder(augmented_message, generator_polynomial) == 0):
            return transmitted_message // 2 errors
```

3.3 Testing

In addition, we tested our code for every possible error and made sure that it actually works as well. Our testing code is as follows -

```
tester():
    for l from 1 to 20:
        generate all bitstrings of length l
        for bitstring in sequence of bitstrings:
            msg <- bitstring
            intended_msg <- encode_crc(msg, generator_polynomial)
```

```
for i, j in length(intended_msg):
    augmented_msg <- intended_msg with i and j bits flipped
    assert(decode(augmented_msg) == intended_msg)
```

In addition, we tested that there is one and only one valid message corresponding to a corrupted message.

4 Some Finer Details

The bits - 0 and 1 are being transmitted at different frequencies which are 4000Hz and 6000Hz respectively. The preamble is being transmitted at 8000Hz. The sender sends a particular bit for 0.3 seconds. The receiver detects frequencies in every interval of 0.05 seconds. So, basically the receiver expects that the window length of a particular frequency will be 6, for it to decide on the bit which was transmitted. Through such a mechanism, we ensure that the synchronization delay is compensated, and we receive the correct bitstring. We stop the receiving side, when we receive three '?' (which do not correspond to the frequencies of 0 and 1).

5 Additional Changes

We have made the following changes after some testing to ensure a robust system and to improve efficiency:

1. Instead of sending one bit (having two different frequency values), for efficiency, we clump up 4 bits as one (having 16 different frequency values for 16 different combinations of 4 bits), effectively reducing our bit duration by 4
2. Also, the preamble contains the length of the message too, since we need to know when to start recording and stop recording

6 Instructions to Run the Code

6.1 Sender

In your terminal, run `python sender.py` (or `python3`). Enter the two real numbers a, b and the message to be transmitted.

6.2 Receiver

In your terminal, run `python receiver.py` (or `python3`).