Lab 9: Real-time Implementation of Frequency-shift Keying (FSK)

EE 352 DSP Laboratory

Lab Goals

- Understand the basics of frequency-shift keying (FSK).
- Implementation of FSK Transmitter and Receiver on TMS320C5515 DSP Board.
- In real-time, continuously transmit and receive data (characters) between two 5515 kits.

1 Introduction

1.1 Overview of frequency-shift keying

Frequency-shift keying (FSK) is a frequency modulation scheme in which digital information is transmitted through discrete frequency changes of a carrier signal. This technology is used for communication systems such as amateur radio, caller ID and emergency broadcasts. The simplest FSK is binary FSK (BFSK). BFSK uses a pair of discrete frequencies to transmit binary (0s and 1s) information. With this scheme, the `1` is called the mark frequency and the `0` is called the space frequency.

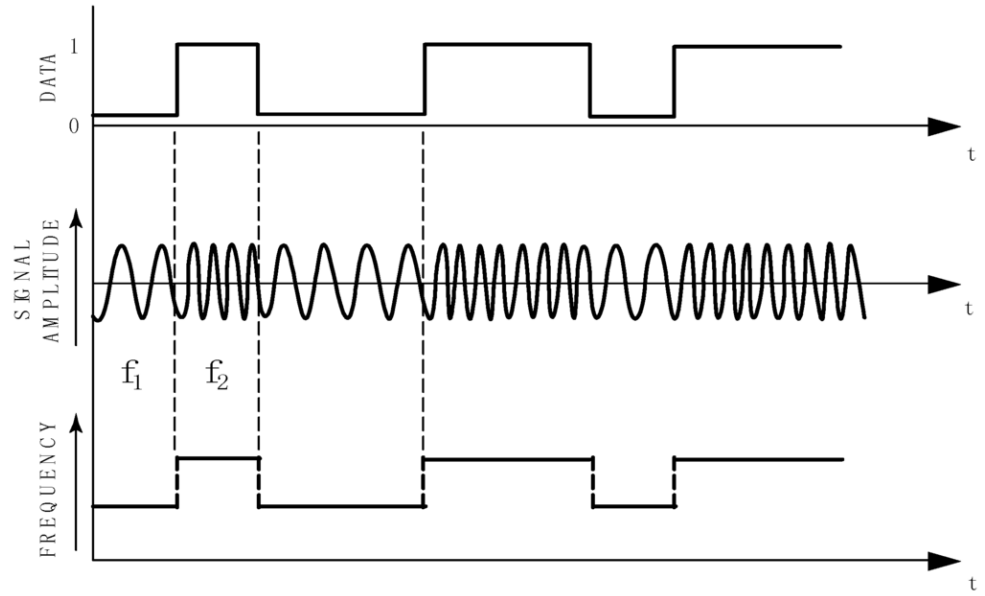

Figure 1: Binary Frequency Shift Keying.

1.2 Overview of Line Coding

In telecommunication, a line code is used for transmitting a digital signal down a line. It is often used for digital data transport. Some line codes are digital baseband modulation or digital baseband transmission methods, and these are baseband line codes that are used when the line can carry DC components.

Line coding consists of representing the digital signal to be transported, by a waveform that is appropriate for the specific properties of the physical channel (and of the receiving equipment). The pattern of voltage, current or photons used to represent the digital data on a transmission link is called line encoding.

In our transmission process we will be using Return-to-Zero coding. Return-to-Zero (RZ or RTZ) describes a line code used in telecommunications signals in which the signal drops (returns) to zero between each pulse. This takes place even if a number of consecutive 0s or 1s occur in the signal. The signal is self-clocking. This means that a separate clock does not need to be sent alongside the signal. It suffers from using twice the bandwidth to achieve the same data-rate as compared to non-return-to-zero format.

For better understanding, go to `https://en.wikipedia.org/wiki/Line_code`

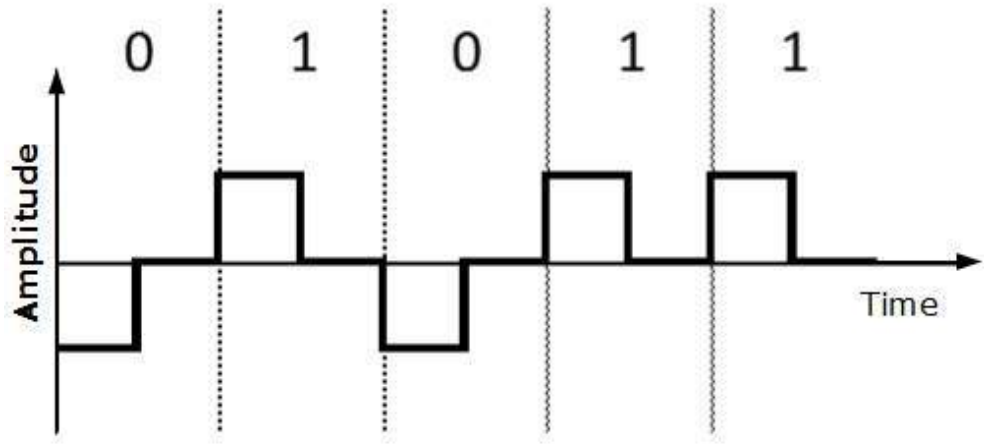

Figure 2: Return to zero Line coding.

2 Implementing a *bit* transmitter and receiver pair using TMS320C5515

Learning objectives for the section

After reading this section, you must be able to,

- successfully set-up a transmitter and receiver using 5515 kits.
- send streams of bits over a wired channel after modulation and demodulate them at the receiver to decode the bits.

2.1 Understanding the bit transmitter

- Create a new project in CCS and copy the `bit_transmitter.c` file that has been provided.
- In the transmitter, we are modulating each bit into square waveforms, which are to be transmitted over the line.
- We are modulating each `bit 1` to a 250 Hz square wave and each `bit 0` to a 500 Hz square wave.
- We are using two functions in our code, `send_one()` and `send_zero()` for the bit transmissions. These functions will take care of the entire transmission process.
- The basic transmission process that we are using is to transmit six cycles of 250 Hz for transmitting a `bit 1` and six cycles of 500 Hz for transmitting a `bit 0`.
- In order for the receiver to know where our data is present in the bit stream, we will be sending a `preamble` before our bit stream. Here the preamble used is the pattern `11010`.

2.2 Learning Checkpoints

Understand the working of the bit transmitter code and explain it to your TA.

2.3 Understanding the bit receiver

- Create a new project in CCS and copy the `bit_receiver.c` file that has been provided.
- For perfect reception and demodulation you need to take care of two things; one is setting the thresholds for `bit 1` and `bit 0` receptions, and the second is to implement the synchronization between the data being sent and received.
- For setting thresholds, you need to send a desired bit continuously over the line and check what value you are receiving in the `LEFT_DATA` or `RIGHT_DATA` in the provided code. This will help you decide the proper threshold for both the bits. We have taken them to be +4500 and -4500, but they can vary in your case, based on the kit's output voltage level and the channel attenuation. You need to first identify these values, in case the values provided by us don't work.
- Secondly for getting the proper synchronization in between the bits, the receiver should be able to properly know the transitions while going from one bit level to other. This is being taken care by our RZ encoding scheme.
- In the receiver code, we will be using two functions i.e. `detect_one()` and `detect_zero()`. The role of these functions is to find the values at the mid points of the high transitions i.e. the peak values of the waveforms. These functions will return a count of the number of peaks received, and based on this count the receiver will decide whether it received a bit `1` or a bit `0`.
- For data extraction, we are using a 30 bit buffer. Whenever the buffer gets full, the receiver will search for the preamble in the buffer and extract the data present after it.

2.4 Learning Checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

1. Understand the function `detect_one()` and make a similar function `detect_zero()`. Keep in mind that the board has an inverter at its output. So whatever data that you sent was actually inverted.
2. Extract the data received in the `data_received` buffer and print it to the console.
3. What is the `bit rate` achieved by your system?

3 Implementing a Character transmitter and Receiver using TMS320C5515

Learning objectives for the section

After reading this section, you must be able to,

- succesfully set-up a transmitter and receiver for data transmission using 5515 kits.
- send any data like characters via the channel to your receiver using 5515 kits.

3.1 Understanding the character transmitter

- Create a new project in CCS and copy the `char_transmitter.c` file that has been provided.
- Before starting with this part, you can use the bit transmission process to verify whether you have established a proper communication channel. Once you are sure of this, you can proceed further.
- The transmission process is the same as for bit transmissions. The only difference between character transmission and bit transmission is that in bit transmissions we were continuously transmitting same data over the channel. Due to this, the constraint on the receiver buffer size was not a problem. In character transmission if we use the same approach then for sending a large amount of data we will need a large receiver buffer which is not present with us. In character transmission we will try to solve this

issue so that we can transmit and receive as much data as we like without any constraint on the buffer size.

- The functions used are similar to that in the bit transmitter. For transmitting a character, we will first convert the character to its ASCII binary representation. For transmitting each character we will use a 16 bit data packet. The first 8 bits will be our preamble `00011010`, and the next 8 bits will contain the ASCII value of our character in binary form.

3.2 Learning Checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

1. Explain the working of the code to your TA.
2. Use the pointer `bin_data_pointer` to transmit the data. You can make use of the `transmit()` function provided for this.

3.3 Understanding the character receiver

- Create a new project in CCS and copy the `char_receiver.c` file that has been provided.
- In the receiver we will only be using one 8 bit buffer `data_received[]` for our reception.
- When a new bit enters, we check if the byte formed by previous seven bits and this bit matches the preamble. If it does, then we will use this buffer to store the next 8 bits as our data bits.
- Once we have extracted the data, we will convert it back to the ASCII value of the character.

3.4 Learning Checkpoints

All the following tasks need to be shown to your TA to get full credit for this lab session.

1. Make a function `check_pattern()` which will check for the preamble each time we receive a bit. You can use the variable `no_match_count` that we have defined. This variable globally stores the number of non-matches to preamble till now. It helps us find the index of the `data_received` buffer from where you can start to find a match.

   Suppose 1st input bit arrives. It will be stored in `data_received[0]`, `no_match_count` should help me compare `data_received[1:7]+data_received[0]` with preamble. If no match then with next input compare `data_received[2:7]+data_received[0:1]` with preamble, and so on.

   If match is found, `no_match_count` will be reset to its initial value and the `data_next` will be set to 1. `data_next = 1` means that the next 8 bits will be the data.

   Take proper measures to receive this data and again start the cycle for next reception by properly reinitializing the value in `no_match_count`.
2. Send a *static* message from one kit (host PC) and display it by receiving the data on another kit (client PC). This has to happen continuously. Show this to your TA.
3. What is the `character rate` achieved by your system?

4 Additional Checkpoint for extra credit

Can you improve the bit rate of this system? Propose some changes that can help to improve the bit rate. Try to implement them.

Is it possible to send *dynamic* messages between the kits? It will be similar to chatting across PCs using DSP kit. Can you suggest some methods for this? Try to implement it.