1. Encode the bit-pattern 1010000101 using the following digital encoding schemes. Be sure to write down any assumptions you need to make:

   a. NRZ-I
      Need to assume: previous state of signal (assume negative) and meaning of transition (in this case, assume a “0” bit is represented by no transition and a “1” bit is represented by a transition)

   b. Manchester
      Need to assume: previous state of signal (assume negative)

   c. AMI
      Need to assume: polarity of last pulse (assume negative)

   d. B8ZS
      Need to assume: polarity of last pulse (assume negative) and number of consecutive zeroes just before the beginning of the graph (assume 0).

Note that this looks just AMI – in this case, no further substitution is needed.
2. For each of the following issues discuss the limitations of NRZ-L. Explain how the Manchester encoding overcomes these limitations.

a. Asynchrony

For the NRZ-L encoding, given a long string of 1’s or 0’s, there are no transitions, which means that we (the receiver) must rely on time alone to determine where bits begin or end (i.e. by looking at a clock). If our clock does not run at precisely the same rate as the transmitter’s clock then we may misinterpret the signal and lose synchronization with the transmitter. If the level of the signal changes (called a *transition*) then that is an event that we can recognize and use to re-synchronize ourselves.

Manchester encoding solves this problem by ensuring that there is a transition in the middle of every bit. Thus, for any sequence of bits, we are guaranteed to have transitions that can be recognized, and thus can be used to synchronize with the transmitter in every bit-interval.

b. DC component

The DC component is the component of a signal with a frequency of 0. You can also think of the DC component as a vertical-shift in the time-domain graph of the signal. We can determine if a signal has a DC component by calculating its average value over time. If its average value is 0, then the signal has no DC component. In the case of NRZ-L, there are possible sequences of bits (e.g. 100100100…) which have an average value which is clearly not zero, and thus NRZ-L has a DC component.

In Manchester encoding, the value of the signal in each bit-interval occupies both polarities – it spends half of its time at the negative polarity and half of its time at the positive polarity. Thus the average value of Manchester encoding over any bit-interval (regardless of the bit value that interval represents) is 0. Thus, Manchester encoding does not have a DC component.
3. What is the highest frequency that can be represented by a digital signal with a bit-rate of 64Kbps? Explain your answer.

If our digital signal represents “frequencies”, then we can reasonably assume that it is representing analog information. Our sampling theorem tells us that:

\[ f \leq \frac{f_s}{2} \]

That is, the digital representation of an analog signal represents no frequency higher than half the sampling frequency.

However, we are not given a sampling frequency here – just a bit-rate. The sampling frequency will be dependent upon the number of bits we use to approximate the level of each sample. We can reasonably assume that at least 1 bit must be used to represent the level of each sample.\(^1\) So that implies a maximum sampling frequency of 64KHz. Thus the maximum frequency that can be represented is 32KHz.

4. A stream of bits is encoded using the ASK method. Only two amplitudes are used – one for the bit value “0” and one for the bit value “1”. The bit-rate of the digital information is 9 Mbps and the carrier frequency is chosen to be 5Mhz. Is this approach practical? Explain why or why not.

We know that the bandwidth of a two-level ASK signal is:

\[ B = (1 + r)R \]

where \( R \) is the bit-rate and \( r \) is the “filter quality” which typically is in between 0 and 1. So the minimum bandwidth occurs when \( r = 0 \). As this point, the bandwidth of the encoded 9Mbps signal would be 9MHz. The “worst case” bandwidth is when \( r = 1 \), at which point the bandwidth would be 18MHz. Since the modulated signal is centered around the carrier frequency, in the worst case, the maximum frequency of the encoded signal would be 14MHz \((5 + 18/2)\) and the minimum would be -4MHz. We cannot have a negative frequency – thus this approach is not practical. The carrier frequency is too low.

**Error Detection**

1. Given the CRC divisor 110011 calculate the CRC check value for the data represented by following sequence of bits: 10010101011.

\(^1\) Of course 1 bit is not a particularly useful size for approximating the level of a signal. As a real-world example, digital telephone systems typically use an 8KHz sampling rate and 8 bits to approximate the level of the signal for an overall data rate of 64Kbps. You should be able to work out on your own the maximum frequency that can be represented in this case.
This is a simple application of the modulo-2 division we did in class. In the interest of consistency, I have revised the solution here to include the padding of the data block with 0-bits as is usually done in any real implementation of CRC. We add enough 0’s to take the place of the remainder (which will be 1 fewer bits than the size of the divisor):

\[
\begin{array}{c|c}
110011 & 100101011100000 \\
110011 & \hline
101100 \\
110011 & \hline
111111 \\
110011 & \hline
011000 \\
000000 & \hline
110001 \\
110011 & \hline
0010101 \\
000000 & \hline
001010 \\
000000 & \hline
010100 \\
000000 & \hline
101000 \\
110011 & \hline
110110 \\
110011 & \hline
001010 \\
000000 & \hline
01010 \\
\end{array}
\]

Thus, the CRC check value is the remainder of the division, which is 01010.
2. In class we looked at the case of parity which could detect a single-bit error but not a double-bit error.

*My apologies for this question which got truncated in the exercise set that was originally distributed. Please ignore it.*

**Flow Control**

1. A link implements the Sliding Window protocol with an initial (and maximum) window size of 5 frames. Assuming that sequence numbers start with 0 show the state of the windows of the both the source and the destination after the following has happened:

   [Before we answer the parts below, it is worth working out the initial state of the windows. In the diagrams below I will show the source’s window on the left and the destination’s window on the right. Since the maximum window size is 5, then we will need at least sequence numbers 0 through 4. However if we are to later implement any form of error control, we will need sequence numbers 0 through 5, so we’ll use 0 through 5 in this solution. The initial windows would look like:

   0 1 2 3 4 5 0 1 2 3 4 0 1 2 3 4

   that is: the transmitter believes it can send 5 frames and the receiver is prepared to accept 5 frames.]

   a. The source sends frames 0, 1, 2 to the destination but they have not been received yet.

      In this case we would expect the source to remove frames 0, 1 and 2 from its window. The destination is unchanged as it has not received any frames yet.

      0 1 2 3 4 5 0 1 2 3 4 0 1 2 3 4

   b. Frames 0, 1, 2 are received by the destination.

      Now we expect the destination to shrink its window, removing frames 0, 1 and 2. Nothing has changed on the source, so its window remains the same:

      0 1 2 3 4 5 0 1 2 3 4 0 1 2 3 4
c. The destination acknowledges frames 0 and 1 and the acknowledgement is received by the source.

As the destination acknowledges frames, it extends its window to reflect that it is ready to receive more frames. When the source receives acknowledgements it expands its window to reflect that it should be able to send more frames to the destination.

```
0 1 2 3 4 5 0 1 2 3 4
```

```
0 1 2 3 4 5 0 1 2 3 4
```

d. The destination acknowledges frame 2 and the acknowledgement is received by the source.

This is the same case that we saw in part c. Both destination and source grow their windows to reflect that another frame can be sent.

```
0 1 2 3 4 5 0 1 2 3 4
```

```
0 1 2 3 4 5 0 1 2 3 4
```

2. Your company is designing a file transfer protocol to send large files from a file server to a client. The communication is mostly in one direction (i.e. data is sent to the client) and the client is assumed to be a modern PC with a large amount of processing power. For the flow control component you have the choice between a protocol that implements a stop-and-wait algorithm and a protocol that implements a sliding-window algorithm with a maximum window size of 16 frames. Which would be the better choice and why?

There are several points here to consider:

- The communication is mostly in one direction
- The communication consists of “large files” – so there is a lot of data to transmit
- Since the data comes from “large files” we would expect that large amounts of data would be ready at once (rather than having to wait for data to be generated).
- The equipment involved has “a large amount of processing power” so we can ignore processing requirements of the schemes

Given these properties it seems clear that the better choice would be the sliding window protocol, where we could send 16 frames without having to wait for an acknowledgement, whereas we could only send one frame at a time with the stop-and-wait approach.

**Error Control**

1. Explain how the Stop-and-Wait-ARQ error control method recovers from each of the following failures:
a. Damaged data frame

If a frame is damaged, that means that it is received but the FCS (frame check sequence) does not match when we apply our error detection algorithm. Thus the receiver knows that an error has occurred. The receiver sends an ACK frame to the transmitter with the sequence number of the damaged frame. At this point, the transmitter will attempt to retransmit the damaged frame.

b. Lost ACK frame

If a frame is lost then it is never received and there is no immediate indication that it has not been received. After the transmitter sends a data frame, it starts a timer. If a predetermined amount of time elapses without receiving an ACK frame from the receiver, then the transmitter will generate a time-out. In this case the transmitter attempts to recover by either re-sending the unacknowledged data frame or by sending a special frame (e.g. ACK) to indicate to the receiver that no acknowledgement has been received.

2. (Hard) Your company is designing a real-time protocol for sending audio and video from one computer to another. The audio and video is played back in real-time as it is received by the destination computer. What would be an appropriate error control algorithm for such a setup and why?

To come up with an answer to this problem we need to carefully consider both what error control accomplishes for us as well as the kind of communication that is being performed. In this case, the information being transferred is audio and video information, in particular, real-time information. By “real-time” we can infer that the information is being generated/processed in “real-time” – i.e. 2 seconds of video is played back in 2 seconds. Consider an example like a cell phone conversation as “real-time” communication – the data is generated (by someone speaking) in real-time and the data is decoded and processed (by generating sounds on the phone) in real-time as the data is received.

Consider the audio case alone for a moment as we have considered this case in the lectures. CD audio (for example) has a sampling frequency of 44.1KHz and a 16-bit sample size. CD audio is in stereo (meaning that there are 2 channels) so a bit-rate of about 1.41Mbps is required in order to transmit CD audio in real-time. With this example in mind, consider the case where we use a 1.41Mbps channel to transmit CD audio in real-time. What happens when we encounter an error? Our error control algorithm must notify the source that a frame needs to be retransmitted. This means that we would need to wait until the rejection notice is sent, received, processed and the damaged frame is resent. However, while this is going on, the person using the receiver is still expecting audio to play out of their speakers (or is waiting to hear the next bit of conversation from their cell phone). Since the frame has been delayed, we will likely receive the correct frame too late to play it. The result will sound like a skip or a pause in the audio playback. So
the quality of our output is sacrificed. Not only do we sacrifice the quality of our output, but we also fall behind in time due to the delay in receiving the correct frame. That is, frames after the damaged frame will be delayed even if they are received correctly.

Because of these issues many real-time communication schemes do not implement error control at all. Sometimes it is better to let the error go (and in this case, create a brief burst of static) and hope for the best in the future rather than fall behind.

**Error Correction**

1. Given a frame size of 128 bits, how many check bits would be required to detect and correct any single-bit error in the frame?

We came up with a general relationship between the number of bits required to correct a single-bit error in the lecture:

\[ 2^r \geq m + r + 1 \]

where \( m \) is the number of bits in the frame, and \( r \) is the number of check bits. If we substitute the frame size we get:

\[ 2^r \geq 128 + r + 1 \]
\[ 2^r - r \geq 129 \]

This is a non-trivial relation to solve, but the numbers are small and simple so we can probably solve it by simple trial and error. 128 is \( 2^7 \), so we can substitute 7 and see that it will not work (\( 2^7 - 7 = 121 \)). If we try \( r = 8 \), we get:

\[ 2^8 - 8 = 256 - 8 = 248 \]

which is greater than 129, so we need 8 check bits to correct a single-bit error in a frame of 128 bits.

2. (Hard) Given the same frame size of 128 bits, how many check bits would be required to detect and correct any single-bit or double-bit (i.e. two bits in the frame are wrong) error? Generalize your answer to determine the relation between \( m \), the number of bits in the frame and \( r \), the number of check bits.

To solve this, you need to think back to the principles we used to derive the number of bits required to correct a single-bit error:
We’ll solve this question “backwards” by deriving the general form first and using that to solve the specific case. The left-hand side of the inequality will be the same – \( r \) bits still represents \( 2^r \) possible states. What are the possible states that we need to represent?

- One bit is incorrect
- Two bits are incorrect
- No bits are incorrect (i.e. no error)

If we add \( r \) check bits to the frame then the total size of the frame will be \((m + r)\). We know from our previous discussion that there are \((m + r)\) possible cases if exactly one bit is incorrect (remember that the check bits could be wrong as well, so we need to count them as part of our discussion). How many possible cases are there where exactly two bits are incorrect? Well the for the first possible incorrect bit, there are \((m + r)\) possible choices. For the second incorrect bit there are \((m + r – 1)\) possible choices (we exclude the bit we chose for the first incorrect bit). If you list all such possibilities, you will notice that each possible pair of bits will show up in that list twice – in reverse order. Of course, which is the “first” incorrect bit and which is the “second” incorrect bit is arbitrary – in other words, it doesn’t matter. So the total number of such double-bit errors is:

\[
\frac{(m + r)(m + r - 1)}{2}
\]

If you have some experience with combinatorics, you would know that there is already a formula for the number of possible ways to choose \( k \) objects from a set of \( n \) objects if we don’t care about the ordering:

\[
\binom{n}{k} = \frac{n!}{(n - k)!k!}
\]

[The left-hand construct is read as “\( n \) choose \( k \).”] So we know the number of possible double-bit error cases is “\( m + r \) choose 2” which would be:

\[
\binom{m + r}{2} = \frac{(m + r)!}{(m + r - 2)!2!} = \frac{(m + r)(m + r - 1)}{2}
\]

So with this result we can determine the total number of possible states:

\[
\frac{(m + r)(m + r - 1)}{2} + (m + r) + 1
\]
That is, the number of ways two bits can be wrong plus the number of ways one bit can be wrong plus the case where there is no error at all. So our general result for correcting any single-bit or double-bit error is:

\[ 2^r \geq \frac{(m+r)(m+r-1)}{2} + (m+r) + 1 \]

So going back to our specific case of a frame size of 128 bits:

\[
\frac{(128 + r)(128 + r - 1)}{2} + (128 + r) + 1 \\
= \frac{128^2 + 256r + r^2 - 128 - r}{2} + 128 + r + 1 \\
= 8129 + \frac{257}{2} r + \frac{1}{2} r^2
\]

which gives us the relation:

\[ 2^r \geq 8129 + \frac{257}{2} r + \frac{1}{2} r^2 \]

Again – this is not easy to solve directly, so we’ll use trial and error. \(2^{13} = 8192\), so we’ll try that first:

\[
8129 + \frac{257}{2} 13 + \frac{1}{2} 13^2 \\
= 9884
\]

So 13, doesn’t work. What about 14? Well, \(2^{14} = 16384\), so:

\[
8129 + \frac{257}{2} 14 + \frac{1}{2} 14^2 \\
= 10026
\]

which works. So to correct any single-bit or double-bit error in 128 bits of data, we would need to add 14 check bits.