THEORY OF OPERATIONS AND REQUIREMENTS:

The modem will FSK, a technique used in 1200-baud modems.

Keying alludes to Morse code-style keying.

As shown in Fig. 1, the FSK scheme transmits sinusoidal tones with 0 and 1 assigned to different frequencies.

Sinusoidal tones are much better suited to transmission over analog phone lines than are the traditional high and low voltages of digital circuits.

![Sinusoidal wave diagram]

The 01 bit patterns create the chirping sound characteristics of modems.
The scheme used to translate the audio input into a bit stream is illustrated in the above fig.

One filter passes frequencies in the range that represents a 0 and rejects the I-band frequencies and the other filter does the converse.

The outputs of the filters are sent to detectors, which compute the average value of the signal over the past n samples.

When the energy goes above a threshold value, the appropriate bit is detected.
STUDENTSFOCUS.COM

Figure: Receiving bits in the modem

Name: Modem
Purpose: A fixed band rate frequency keypad modem
Inputs: Analog sound input, reset button, microphone input
Outputs: Analog sound output, LED display
Functions: Transmitter: Sends data from user to device
Receiver: Automatically detects bytes and stores result

Performance: 1200 baud

Manufacturing cost: Dominated by ICs and analog I/O circuits
**System Architecture:**

Fig: Class diagram for the modem

**Time**

Fig: Waveform generation by table look up

- Fig. shows an analog waveform with same points and the code for these samples.

**Table look-up can be combined with interpolation to generate high-resolution waveforms without excessive memory costs.** This is more accurate than oscillators because no feedback is involved.
DATA COMPRESSOR

Our design example for this chapter is a data compressor that takes in data with a constant number of bits per data element and puts out a compressed data stream in which the data is encoded in variable-length symbols.

Requirements and Algorithm:

We use the Huffman coding technique, which is introduced in Application. We require some understanding of how our compression code fits into a larger system. Figure shows a collaboration diagram for the data compression process.

The data compressor takes in a sequence of input symbols and then produces a stream of output symbols. Assume for simplicity that the input symbols are one byte in length. The output symbols are variable length, some have to choose a format in which to deliver the output data. Delivering each coded symbol separately is tedious, since we would have to supply the length of each symbol and use external code to pack them into words. On the other hand, bit-by-bit delivery is almost certainly too slow. Therefore, we will rely on the data compressor to pack the coded symbols into an array. There is not a one-to-one relationship between the input and output symbols, and we may have to wait for several input symbols before a packed output word comes out.

Application Example:

Huffman coding for text compression

Figure UML collaboration diagram for the data compressor

Text compression algorithms aim at statistical reductions in the volume of data. One commonly used compression algorithm is Huffman coding [Huf52], which makes use of information on the frequency of characters to assign variable-length codes to characters. If shorter bit sequences are used to identify more frequent characters, then the length of the total sequence will be reduced.

In order to be able to decode the incoming bit string, the code characters must have unique prefixes: No code may be a prefix of a longer code for another character. As a simple example of Huffman coding, assume that these characters have the following probabilities $P$ of appearance in a message:

<table>
<thead>
<tr>
<th>Character</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.45</td>
</tr>
<tr>
<td>B</td>
<td>0.24</td>
</tr>
<tr>
<td>C</td>
<td>0.11</td>
</tr>
</tbody>
</table>

We build the code from the bottom up. After sorting the characters by probability, we create a new symbol by adding a bit. We then compute the joint probability of finding either one of those characters and re-sort the table. The result is a tree that we can read top down to find the character codes. The coding tree for our example appears below.
Reading the codes off the tree from the root to the leaves, we obtain the following coding of the characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>D</td>
<td>0001</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
<td>E</td>
<td>0010</td>
</tr>
<tr>
<td>C</td>
<td>0000</td>
<td>F</td>
<td>0011</td>
</tr>
</tbody>
</table>

Once the code has been constructed, which in many applications is done off-line, the codes can be stored in a table for encoding. This makes encoding simple, but clearly the encoded bit rate can vary significantly depending on the input character sequence. On the decoding side, since we do not know a priori the length of a character’s bit sequence, the computation time required to decode a character can vary significantly.

The data compressor as discussed above is not a complete system, but we can create at least a partial requirements list for the module as seen below. We used the abbreviation N/A for not applicable to describe some items that do not make sense for a code module.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data compression module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Code module for Huffman data compression</td>
</tr>
<tr>
<td>Inputs</td>
<td>Encoding table, uncoded byte-size input symbols</td>
</tr>
<tr>
<td>Outputs</td>
<td>Packed compressed output symbols</td>
</tr>
<tr>
<td>Functions</td>
<td>Huffman coding</td>
</tr>
<tr>
<td>Performance</td>
<td>Requires fast performance</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>N/A</td>
</tr>
<tr>
<td>Power</td>
<td>N/A</td>
</tr>
<tr>
<td>Physical size and weight</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Specification:
Let’s refine the description of Figure 3.20 to come up with a more complete specification for our data compression module. That collaboration diagram concentrates on the steady-state behavior of the system. For a fully functional system, we have to provide the following additional behavior:
- We have to be able to provide the compressor with a new symbol table.
We should be able to flush the symbol buffer to cause the system to release all pending symbols that have been partially packed. We may want to do this when we change the symbol table or in the middle of an encoding session to keep a transmitter busy.

A class description for this refined understanding of the requirements on the module is shown in Figure. The class’ buffer and current-bit behaviors keep track of the state of the encoding, and the table attribute provides the current symbol table. The class has three methods as follows:

- **Encode** performs the basic encoding function. It takes in a 1-byte input symbol and returns two values: a boolean showing whether it is returning a full buffer and, if the boolean is true, the full buffer itself.

![Figure: Definition of the Data-compressor class](image)

- **New-symbol-table** installs a new symbol table into the object and throws away the current contents of the internal buffer.

- **Flush** returns the current state of the buffer, including the number of valid bits in the buffer.

We also need to define classes for the data buffer and the symbol table. These classes are shown in Figure. The data-buffer will be used to hold both packed symbols and unpacked ones (such as in the symbol table). It defines the buffer itself and the length of the buffer. We have to define a data type because the longest encoded symbol is longer than an input symbol.

The longest Huffman code for an eight-bit input symbol is 256 bits. (Ending up with a symbol this long happens only when the symbol probabilities have the proper values.) The insert function packs a new symbol into the upper bits of the buffer; it also puts the remaining bits in a new buffer if the current buffer is overflowed.
DEBUGGING CHALLENGERS:

- Logical errors in software can be hard to track down, but
  errors in real-time code can create problems that are even
  harder to diagnose.

- If they are too long, they create very unexpected behavior.

**ALARM CLOCK:**

**REQUIREMENTS:**

- Alarm on/Alarm off
- Set time
- Set alarm
- Hour
- Minute
- Alarm ready

[Diagram of alarm clock front panel]

**Name:** Alarm Clock

**Purpose:** A 24-hour digital clock with a single alarm.

**Inputs:** Six push buttons: Set time, Set alarm, hour, minute, alarm on, alarm off.
Outputs: Faun - digit, clock-style output.

Functions: Default mode: The display shows the current time.
Depress set time button:
The button is held down while hour/minute button are pressed to set time.
Depress set alarm button:
While this button is held down, display shifts to current alarm setting. Depressing hour is similar to setting time.

Fig: Class diagram for the alarm clock.

We have three classes that represent physical elements: lights, fan, and buzzer.
and Speaker * from sound output.
* The BUTTONS* class provides read-only access to the current state
of the buttons.
* The lights* class allows us to
drive the lights.

**However**, to save pins on the
microcontroller, to save pins on the
display, **Light** provides signals for
only one digit.

- **Display**:
  - time: integer
  - alarm: indicator
  - PM indicator
  - set-time()
  - alarm light on
  - PM light on
  - PM light off

- **Buttons**:
  - set time()
  - alarm: indicator
  - PM light on

- **Lights**:
  - alarm: indicator
  - PM light on

- **Speaker**:
  - buzz()

F10: DETAILS OF MVLVL LEVEL CLASS FOR THE
ALARM CLOCK.
* We generate the display by
scanning the digits periodically.
The mechanism class is described in Figure 1.

**MECHANISM**

<table>
<thead>
<tr>
<th>SECONDS: INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM: BOOLEAN</td>
</tr>
<tr>
<td>TENS - HOURS, ONES - HOURS: INTEGER</td>
</tr>
<tr>
<td>TENS - MINUTES, ONES - MINS: INTEGER</td>
</tr>
<tr>
<td>ALARM TENS - HOURS, ALARM - ONES - MINUTES: INTEGER</td>
</tr>
</tbody>
</table>

**UPDATE TIME**: RUNS ONCE PER SECOND

**SCAN KEYBOARD ( )**: RUNS PERIODICALLY

The class keeps track of the current time, the current alarm time, whether the alarm has been turned on, and whether it is currently buzzing.

- The clock shows the time only to the minute, but it keeps internal time to the second.
- The time is kept as discrete digits rather than a single integer to simplify transferring the time to the display.

The class provides two behaviours, both of which run continuously.
STUDENTSFOCUS.COM

SYSTEM ARCHITECTURE:

START

UPDATE SECOND'S CLOCK
WITH ROLLER

UPDATE HH:MM
WITH ROLLER

AM -> PM
ROLLER

PM == TRUE
PM-AM ROLLER

PM == FALSE

TIME > = ALARM AND ALARM ON?

DISPLAY SET TIME (CURRENT TIME)

ALARM + BUZZER (TRUE)

END

STATE DIAGRAM FOR UPDATE TIME

At end of interrupt, do

- Check if time needs to be updated
- Update time

Check if time needs to be updated
- Update time
- Display new time

- Alarm and buzzer

END

STUDENTSFOCUS.COM
**Diagram Description:**

- **Start**
  - Update seconds clock with rollover
  - Rollover? / Activations?

- **If Rollover?**
  - Set-time, set-alarm and minutes
  - Increment time tens with rollover
  - Increment time ones with rollover and AM/PM
  - Save button states for next activation

- **If Not Rollover**
  - Alarm off
  - Set time and set-alarm and hours
  - Alarm ready = false
  - Update time with rollover

**Textual Description:**

The foreground program can read the buttons and execute their commands.

Since buttons are changed at a relatively slow rate, it makes no sense to add the hw
required to connect the buttons to interrupts.