I^{2}C Temperature sensor

In the following labs you will read and display data from an I^{2}C temperature sensor and use a logic analyzer to view and interpret I^{2}C bus activity.

We'll use a USB based logic analyzer which requires you to install driver software. Got to https://www.saleae.com/downloads to download and install logic analyzer software.

Bitwise operators in C/C++

For more compact data, programs can store information in individual bits or groups of bits. Many integrated peripherals and external sensors pack information from multiple sources into a single value. A typical case is device status where individual bits indicate device status. For example, TC74 temperature sensor contains an 8-bit configuration register. When the register is read the value returned reflects the status of the device. Bit 7 tell if the device is in standby mode and bit 6 if new data is available.

Individual bits or groups of bits are referred to by their position. Position numbering starts from the least significant bit (the rightmost bit of a binary number) which is at position 0.

```
Bit pattern    0 0 1 0 1 0 1 1

Bit position   7 6 5 4 3 2 1 0
```

Bitwise operators perform Boolean operations on each bit position of their operands. The operator is applied to each of the bits in one operand and a bit in the same position in the other operand. A bit that is set (=1) is considered true and a bit that is cleared (=0) is considered false.

In the example above when we want to know if bit 3 is set, we can use bitwise operations to test the bit. Testing can be done using bitwise AND operation and second operand where only bit 3 is set. This second operand is often called **mask**. In the result all bits except bit 3 are set to 0 since the result of AND yields zero if either of the operands is zero. In the result

```
Bit position   7 6 5 4 3 2 1 0

Value          0 0 1 0 1 0 1 1   (0x2B)

Mask           0 0 0 0 1 0 0 0   (0x08)

Bitwise AND

Result         0 0 0 0 1 0 0 0
```

Parenthesis are needed because bitwise AND has lower precedence than comparison

C/C++: `if((value & 0x08) != 0) { /* bit 3 is set */ }`

For example, when we read the configuration register of TC74 and want to see if data is available, we can do a bitwise and between the value of configuration register and 0x40 (0100 0000). If the result is non-zero, we know that bit 6 of the config register was set which indicates that a new temperature reading is available.

Operator	Description	Precedence
~	Bitwise NOT	Very high
&	Bitwise AND	Low
^	Bitwise XOR (exclusive or)	Low
|	Bitwise OR (inclusive or)	Low

Note that bitwise operators can be used only on integer types.

Shifting

In addition to bitwise Boolean operators we have operators for shifting bits to left or right. Shifting can be used, for example to multiply or divide integers by powers of two, but the most common use is to place individual bits or groups of bits at specific bit positions. For example, there are 7-bits in an I2C address but when the value is written to the bus, we need one additional bit to tell the slave if the operation is read or write. The R/W bit is the least significant bit of the data so the 7-bit address needs to be shifted to left to make space for R/W bit.

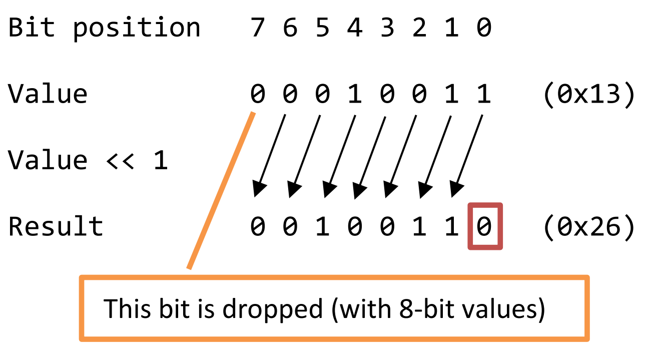
The bitwise shift operators are << and >>. Note that C++ streams overload the operators and use them for input and output. When the operators are applied on built-in integer types they do bitwise shifting. The operators shift the operand on the left side of the shift operator. The second operand determines how many positions to shift.

The address is shifted one position to left. The “empty” bit is set to zero.

Bitwise OR sets the least significant bit to 0 or 1.

```
/* Write Address and RW bit to data register */
Chip_I2CM_WriteByte(pI2C, (xfer->slaveAddr << 1) | (xfer->txSz == 0)); // original
```

If transmit size is zero, which means that we are going to read, the comparison is true (=1). Comparison operator yields 0 or 1 so it is safe use the result with bitwise or



```
Bit position   7 6 5 4 3 2 1 0

Value          0 0 0 1 0 0 1 1   (0x13)

Value << 1

Result         0 0 1 0 0 1 1 0   (0x26)
```

Shift and mask

Shifting and bitwise AND is often called *shift and mask*. Shift and mask is used to extract a value stored in a group of bits that does start at bit number 0. For example, LPC1549 AD converter data register (32 bits) contains the following fields:

Bit	Description
3:0	Reserved. The value read from a reserved bit is not defined.
15:4	This field contains the 12-bit A/D conversion result from the last conversion performed on this channel.
17:16	Threshold Range Comparison result.
19:18	Threshold Crossing Comparison result.
25:20	Reserved. The value read from a reserved bit is not defined.
29:26	This bit will be set to a 1 if a new conversion on this channel completes and overwrites the previous contents of the RESULT field before it has been read.
30	This bit will be set to a 1 if a new conversion on this channel completes and overwrites the previous contents of the RESULT field before it has been read.
31	This bit is set to 1 when an A/D conversion on this channel completes.

When we want to read the conversion result, we need to move it to start from position 0 and to remove the status bits after the conversion result. This can be done by shifting the value to right by four bits and then masking the unwanted bits with bitwise AND.

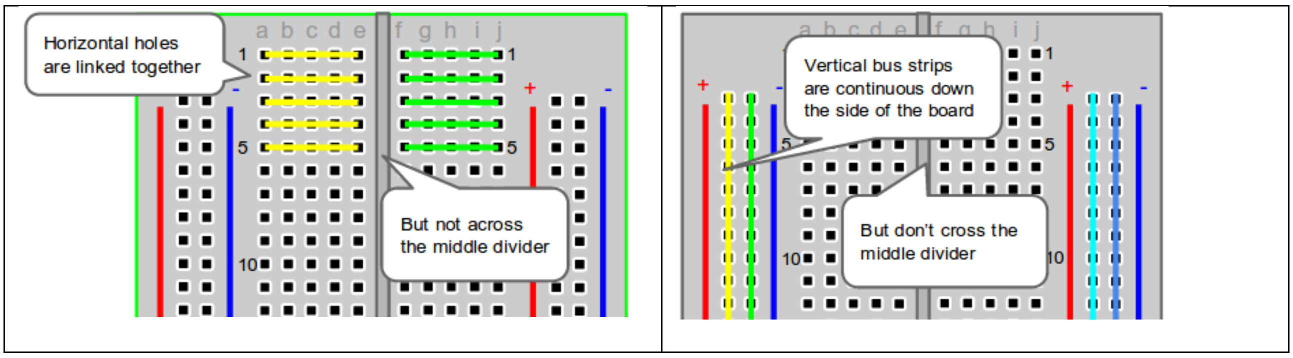
```
/* Macro for getting the ADC data value */
#define ADC_DR_RESULT(n)  ((((n) >> 4) & 0xFFF))
```

Shift to right by four positions

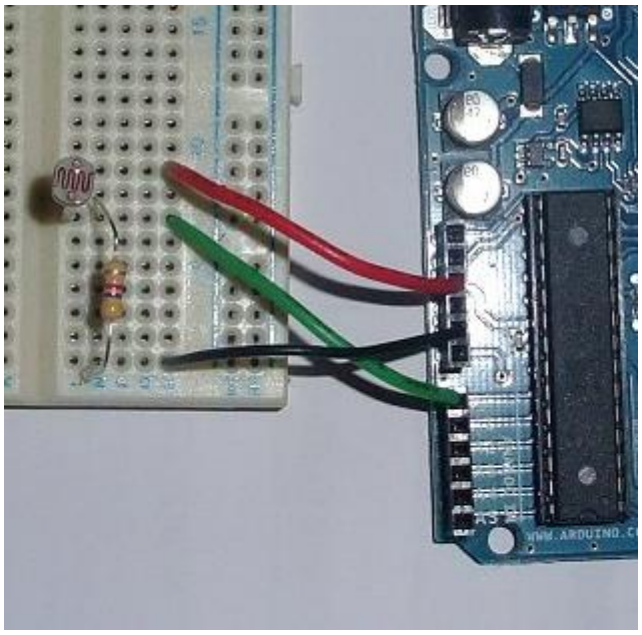
This mask keeps 12 least significant bits. The lowest 12 bits are all ones and the rest are zeros

Solderless breadboard

In the following exercises, you need to wire a temperature sensor to your LPCXpresso. For wiring we use solderless breadboard and solid wires.


Horizontal holes are linked together
But not across the middle divider
Vertical bus strips are continuous down the side of the board
But don't cross the middle divider

Breadboard strip connections



Example of breadboard wiring

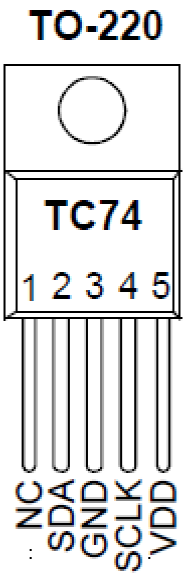
In addition to the breadboard you need the following parts (ask instructor for the parts):

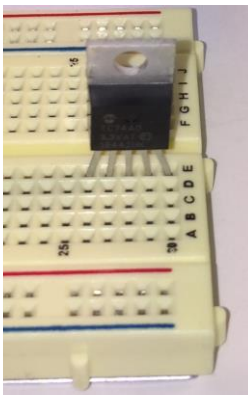
- 4 pieces of solid wire (5-7 cm each) for exercises 1 and 2
    - Strip 5 mm of insulation from both end of the wire
- TC74 temperature sensor
- Logic analyzer and dual headed pins for Exercise 2
    - Cold spray for final testing of Exercise 2
- 3 leds and 3 resistors and additional 4 pieces of solid wire for exercise 3

Logic analyzers are only available in the class. You may not borrow the analyzers!

Exercise 1

Connect the temperature sensor to the breadboard. You need to bend the legs of the device a bit to make them align to the holes. The legs break easily so bend them carefully. You can use the unsoldered pins of an **unplugged** LPCXpresso board to align the pins at correct distance.





TC74 pin numbering

Unplug your LPCXpresso before wiring the sensor.

Wire the temperature sensor to your your board:

- TC74 SDA (pin 2) to D14 on LPCXpresso
- TC74 SCL (pin 4) to D15 on LPCXpresso
- TC74 GND to GND on LPCXpresso (located near D14)
- TC74 VDD to AVDD on LPCXpresso (located between GND and D14)

Study example **periph_i2cm_polling** in lpcopen_2_20_lpcxpresso_nxp_lpcxpresso_1549.zip located in C:\nxp\MCUXpressoIDE_10.1.1_606\ide\Examples\LPCOpen (if you installed in default directory).

Study temperature sensor data sheet TC74_datasheet_21462D.pdf (in the workspace). **Sensor address can vary and you need to check the markings on the package to find I2C bus address of the device.** See "Package Marking Codes" on data sheet page 9 and markings on the front of your sensor. Find the minimum and maximum clock rates from the data sheet. Use a clock rate that is roughly at the middle of the available range.

Write a program that:

- Polls SW3
- If SW3 is pressed program reads the status of the device and **reads temperature only if status indicates that device is ready**.

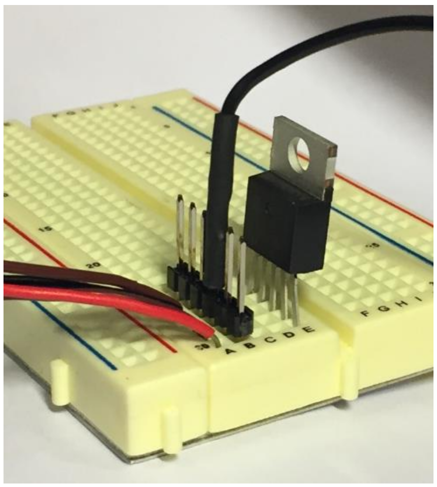
Status can be read using command code 0x01. If reading the status succeeds and bit 6 of device status is set (=1) then read the temperature. Temperature is read using command code 0x00.

Important: Due to a bug in NXP I2C library you need to add a little delay after each transaction. Without the delay the next transaction may start before hardware is ready for it and cause incorrect operation.

Print the values that are read in hexadecimal format using debug UART. Program must clearly indicate the following:

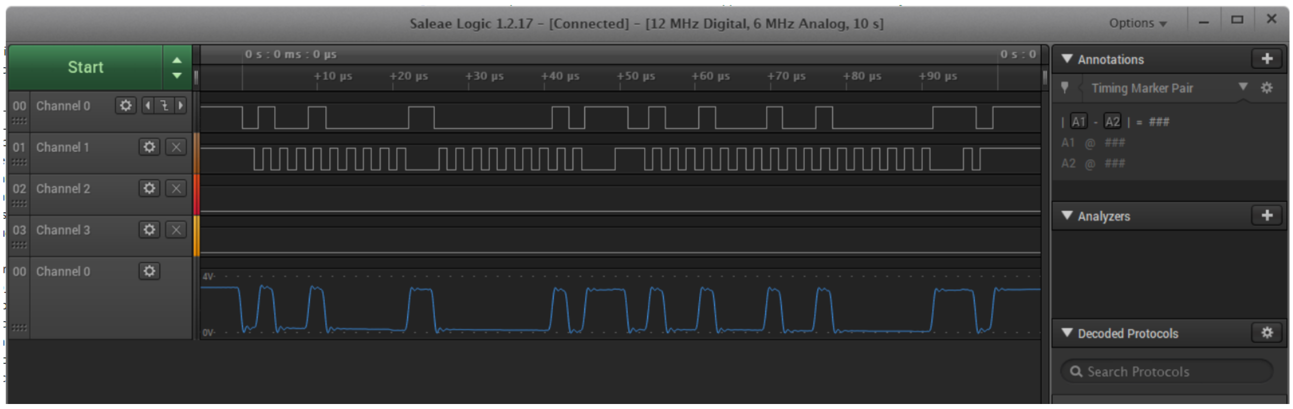
- Did I2C transaction succeed or not
  - Print status only if status read transaction was successful.
- Read temperature only if status read succeeded and status indicates that data is ready
  - Print error message if reading failed
  - Print temperature only if transaction was successful.

Connect logic analyzer to the breadboard. Ask instructor for analyzer and dual headed pins for connecting the analyzer probes. Connect ground probe of channel 0 (black wire with black end) to ground. Connect Channel 0 probe (Black wire, white end that reads ‘0’) to SDA. Connect channel 1 probe (Brown wire, white end that reads ‘1’) to SCL.

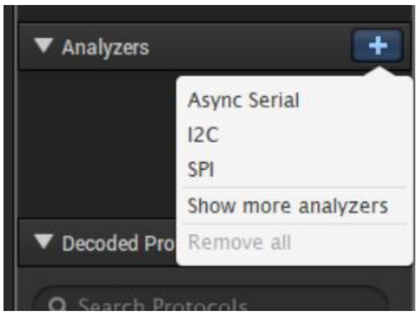


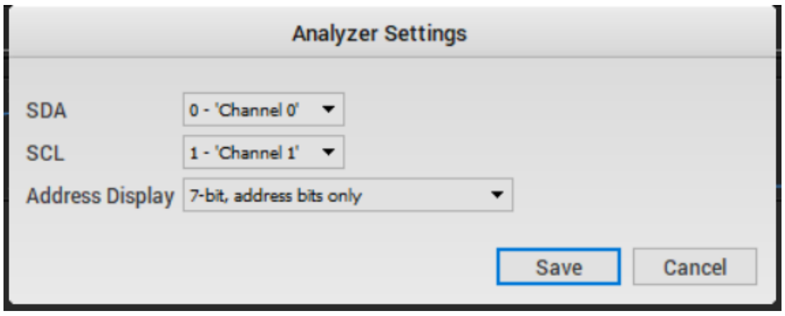
On the left:
How to use dual headed pins to connect logic analyzer. Picture shows only ground probe connected. Channel 0 and 1 probes are connected in the similar way.

Setup logic analyzer to start capture on the falling edge of channel 0. Start debugging your program. When your program is running start capture in the logic analyzer and then press SW3. The capture you get should look something like the following:



The waveforms show what goes on in the I2C bus. To make analyzing easier we can use a built-in protocol analyzer to decode bus transactions in to more human readable form. Click on plus sign next to analyzers and add an I2C analyzer.





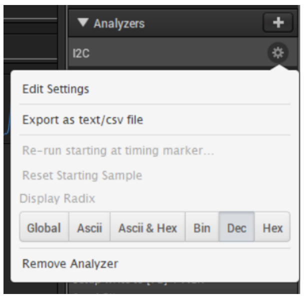
Analyzer Settings

SDA 0 - 'Channel 0'

SCL 1 - 'Channel 1'

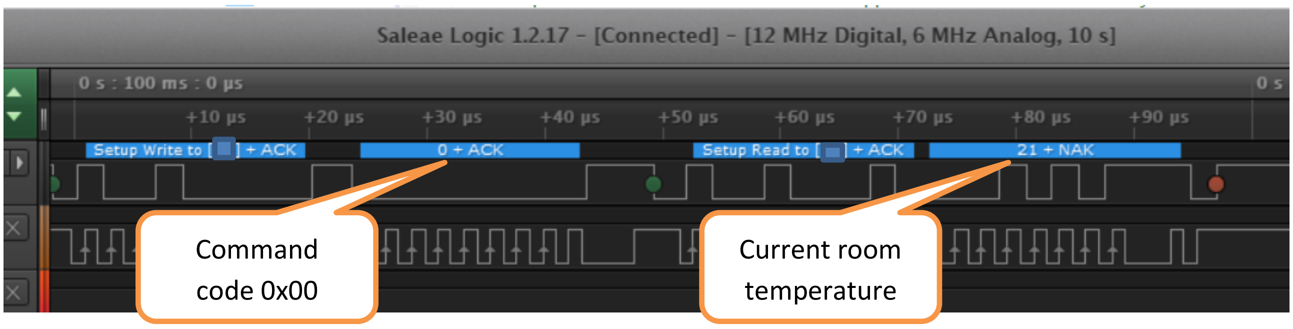
Address Display 7-bit, address bits only

Check that analyzer channel settings match your wiring. Change address display to 7-bit address.



Click on analyzer setting button and change radix to “Dec”.

Find a transaction where you send command code 0. You should see current room temperature in the decoded data. (Device addresses have been hidden in the screenshot below.)



Screenshot 1: Save a screenshot to show instructor that you have completed this step.

Change your program to use some other address (which does not exist…). Run your program again and capture bus activity.

Screenshot 2: Save a screenshot of failed transaction to show instructor that you have completed this step.

Change your program to display the temperature in signed decimal number. See data sheet for information on encoding. Hint: C/C++ variables can be signed or unsigned and you must find a suitable type to which to convert the value.

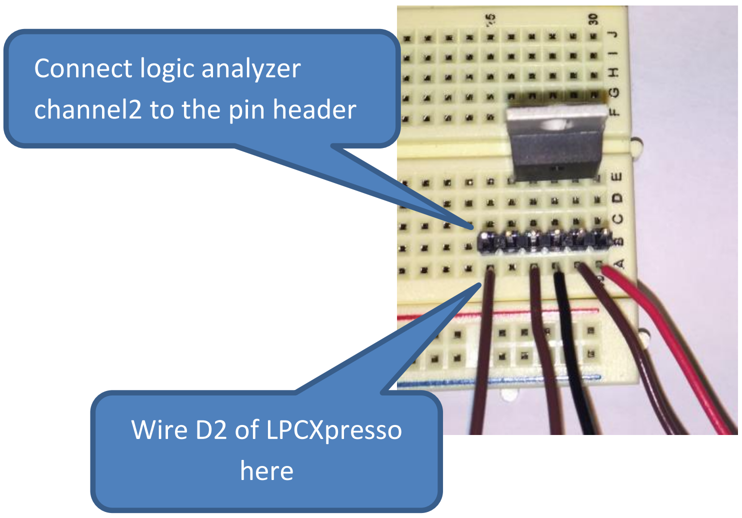
Final test: Test that your program displays same values both on the ITM console and analyzer capture. Then ask instructor for cold spray to cool your sensor below zero centigrade to test that negative temperature values are displayed correctly.

Exercise 2

Change your program so that if SW3 is not pressed for 5 seconds then program puts the sensor in standby mode. When SW3 is pressed the sensor is woken up and temperature is read and printed once through debug UART in JSON format.

When SW3 is pressed the program reads the device status. If the device is in standby mode it is first set to normal mode. **Program must check that device is in normal mode and that data is ready before reading and printing temperature.**

Final test: SW3 is connected to pin D1 of LPCXpresso. Connect logic analyzer channel 2 to the pin to get SW3 state on the analyzer window so that your timing and operation can be verified.


Connect logic analyzer channel2 to the pin header
Wire D2 of LPCXpresso here

Wire D1 to the "unused" pin of dual pin header and connect Channel 2 probe to the pin

JSON message specification

In this exercise the JSON message contains only one type of data: numbers. The message contains three key value pairs as shown in the example below:

```
{
    "samplenr": 1,
    "timestamp": 25007,
    "temperature": 22
}
```

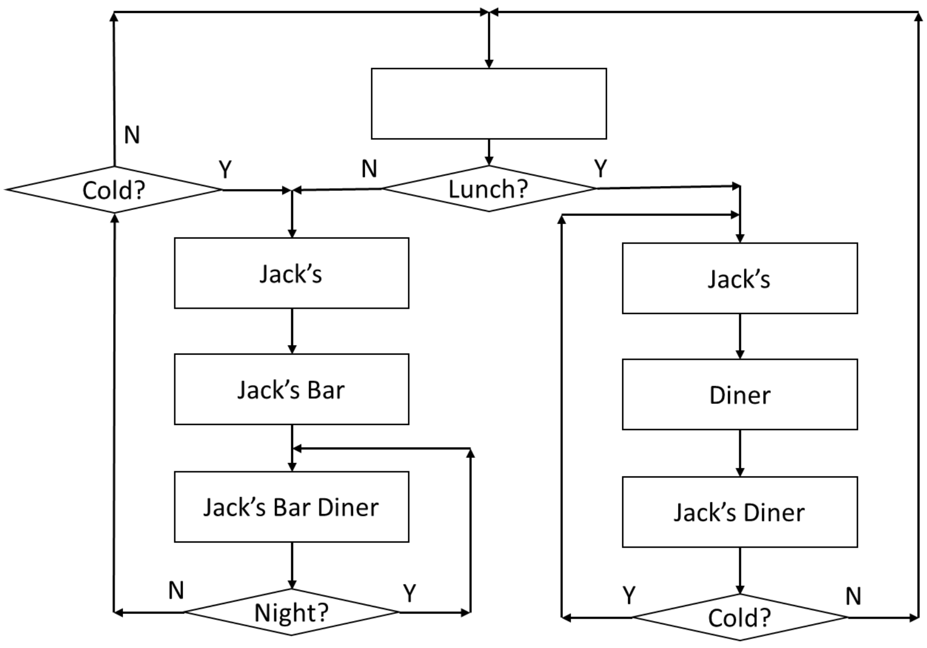
Samplenr is a counter that starts from one and is incremented after each measurement. Timestamp is number of milliseconds since the device was started. Timestamp is taken at the time of measurement. Temperature is the measured temperature.

For more information about JSON see: https://www.w3schools.com/js/js_json_objects.asp

Exercise 3

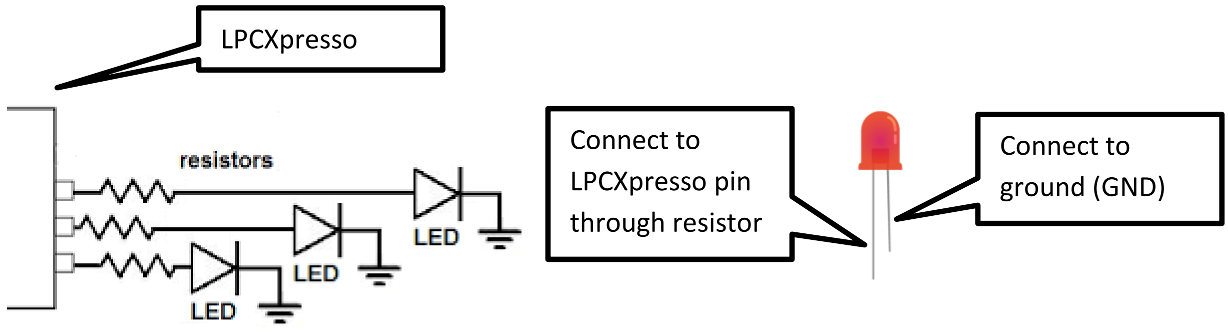
In this exercise we will use three external leds to represent three parts of a neon sign. The controllable parts read: *Jack's*, *Diner*, and *Bar*. You need to write **a state machine-based software** to control the sign's flashing sequence. The flashing sequence is controlled by three inputs: two buttons and the temperature sensor. When temperature sensor indicates that it is cold then at least one part of the sign must be on at all times to prevent frost on the sign. For easy testing select the temperature limit so that you can go over the limit by warming the sensor in your hand. The first button tells if Jack's is serving lunch. The second button puts the sign in night mode where the sign is fully on all the time.

Draw a UML-state machine of the neon sign controller and implement it. The following flow chart shows how the controller should work. The duration of each box in the chart is two seconds. Your state machine does not need to replicate the flow chart (to have the same number of states) but it needs to implement the same behavior.



Use LPCXpresso pins D8, D9, and D10 for the leds.

You need a current limiting resistor for each led to prevent damage to microcontroller pins or leds. Ask instructor for leds and resistors.


LPCXpresso
resistors
LED
LED
LED
Connect to LPCXpresso pin through resistor
Connect to ground (GND)