1 Objective

$\text{I}^2\text{C}$ is a popular, industry-standard serial bus for communications between integrated circuits. Many systems, including most modern PCs, use the $\text{I}^2\text{C}$ bus. Unfortunately, most modern PCs do not make their $\text{I}^2\text{C}$ bus(es) accessible to the end user. This lab will create an $\text{I}^2\text{C}$ interface that will be accessible from the PC.

2 Pre-lab Tasks

1. Review the $\text{I}^2\text{C}$ bus specification (available on the lab webpage)
2. Review the Microchip PIC18Fxx2 $\text{I}^2\text{C}$ and USART peripherals
3. Study the advantages and disadvantages of the “high-level” languages (HLL) of C, Java, and Python. Select one of these three HLLs to implement your PC side software. Justify your decision.
4. Install the appropriate tools and libraries for the HLL selected above. (Be sure that you have “legal” authority to use these tools and libraries. Use of pirated software in this lab is tantamount to academic dishonesty and will be dealt with accordingly.)

3 Specification

This task will create code for your MCU that will communicate with the PC via the serial communications port. Nearly all modern PCs run a pre-emptive multi-tasking OS; however, most modern OS-es can not guarantee timely real-time response. This problem is made worse by the use of virtual machines popular with several platform-independent languages. Since your PC or PC+VM cannot guarantee a timely response, you will design your MCU to be an intermediary between your PC and the $\text{I}^2\text{C}$ bus.

![System block diagram](image)

Figure 1: System block diagram

3.1 Hardware design

Your hardware system should contain buffers to handle bursty transfers in a responsive manner. Therefore, your MCU should perform its communication servicing in ISRs and buffer both incoming and outgoing data streams. (No spin-polling on the TX and RX “done” flags will be allowed.)
Write a MCU program that will accept serial data and “shovel” it to the MCU’s \( I^2C \) bus. \( I^2C \) bus “reads” are sent back to the PC via the serial port. All serial port communications are down with no start bits, eight data bits, and one stop bit. Use the following serial communications packet format for writing to the \( I^2C \) bus:

\[
\begin{array}{cccc}
\text{bytes:} & 2 & 1 & N & 1 \\
0x012C & N & \text{DATA} & \text{CSUM} \\
\end{array}
\]

Figure 2: PC-to-MCU \( I^2C \) write request

where \( N \) is the number of bytes to write to the \( I^2C \) bus, \( \text{DATA} \) is the data payload, and \( \text{CSUM} \) is the 8 bit checksum\(^1\) of data payload. The data payload will be written to the \( I^2C \) bus as-is. That means that it is up to the PC code to send appropriate \( I^2C \) data in \( \text{DATA} \), e.g. proper addresses and \( \text{R/W#} \) bit set. The checksum does not include the two byte header or the number of bytes, \( N \).

When your PC software wishes to read the \( I^2C \) bus, it will use the following serial communications packet protocol:

\[
\begin{array}{cccc}
\text{bytes:} & 2 & 1 \\
0x022C & N \\
\end{array}
\]

Figure 3: PC-to-MCU \( I^2C \) read request

where \( N \) is the number of bytes to read from \( I^2C \) bus. At which point, your MCU will respond with

\[
\begin{array}{cccc}
\text{bytes:} & 2 & 1 & N & 1 \\
0x032C & N & \text{DATA} & \text{CSUM} \\
\end{array}
\]

Figure 4: MCU-to-PC response to \( I^2C \) read request

where \( N \) is the number of bytes to read from the \( I^2C \) bus, \( \text{DATA} \) is the data payload, and \( \text{CSUM} \) is the 8 bit checksum of data payload. Again, the checksum byte \( \text{CSUM} \) does not include the header and the number of bytes field. The number of bytes read may be zero. In this case, the checksum will not be present.

Your PC software and MCU do not have to act upon receipt of bad checksums internally, but your PC code should be aware of the bad transfers and provide a mechanism for the calling application to check the checksum status. With this ability, your application can choose whether it wishes to retry the \( I^2C \) transfer again.

Program your MCU to perform this serial-to-\( I^2C \) translation as a fixed-function device. Your \( I^2C \) transmissions should be at the \( I^2C \) standard of 400 kbps. A high serial baud rate will improve your system’s performance, so take steps to have your MCU communicate at the highest possible reliable rate. Therefore, your MCU will need to run with \( V_{dd} = 5.0V \).

### 3.2 PC side software design

Create software on the PC to communicate with your MCU device to allow HLL applications to read from and write to the \( I^2C \) bus. Your software should be a library (in the case of C) or an \texttt{I2CPort} object (in the case of Java and Python). Your code should implement the following methods\(^2\):

```c
void I2CPort.setPortName( string portName )
void I2CPort.setAddress( int deviceAddress )
boolean I2CPort.open()
boolean I2CPort.isOpen()
boolean I2CPort.read( byte[] barray, int offset, int num )
void I2CPort.write( byte[] barray, int offset, int num )
```

\(^1\)The checksum is calculated with the same method as previous labs.

\(^2\)A C language library implementation should prepend “\texttt{i2c}” to each method name with the initial letter capitalized in the function name verb, e.g. the object-oriented (OO) method \texttt{I2CPort.open()} would be \texttt{i2cOpen()}.\)
boolean I2CPort.isChecksumGood()
void I2CPort.close()

The `setPortName` method takes a `portName` argument to tell your PC code which communications port is used by the PC to talk to the MCU device. The valid values of `portName` are implementation (OS, language, and serial communications library) dependent.

The `setAddress` method sets the \(^2\)C device address. This address will take effect on the next read or write. The argument `deviceAddress` is right justified with the least significant bit of this byte equal to least significant bit of \(^2\)C slave address, excluding the write/read bit.

The `open` method opens the PC serial port and does any needed initialization. It returns true if the port is opened successfully.

The `isOpen` method returns true if the \(^2\)C port is open and ready to be used.

The `read` method initiates an \(^2\)C read of `num` bytes into the array `barray` starting at the `offset` byte. It is up to the caller to make sure that the supplied array, offset, and number of bytes to read are consistent. The method returns true if the read was successful.

The `write` method initiates an \(^2\)C write of `num` bytes into the array `barray` starting at the `offset` byte. It is up to the caller to make sure that the supplied array, offset, and number of bytes to read are consistent.

The method `isChecksumGood` returns true if the most recent transfer had a valid checksum.

The `close` method should close/release the serial port so that other applications may access it.

### 3.3 Test bench

Use your I2CPort library/object class to connect the Microchip 24LC515 serial EEPROM contained in your ECE 3724 parts kit to your PC. Write a program that uses your hardware and PC code to write data to the EEPROM and read it back. Be sure that your test bench fully exercises all aspects of your hardware and software. It is your responsibility to prove that your test bench is exhaustive.

### 4 Demonstrations

You must demonstrate the following to the lab TA (not necessarily before you leave lab):

1. Operation of the your EEPROM by running your test bench
2. Display of serial and \(^2\)C transfers on the logic analyzer with bitrates measured for each
3. Explanation of how your test bench is complete and exhaustive

### 5 Deliverables

1. Document showing your design analysis justifying component values and circuit topology. (‘I put it together and it just worked!’ is not acceptable. You must justify your design decisions.) If you are unable to meet any of the design specifications, you must provide justification as to why it is not possible.

2. Schematic capture of your circuit showing all components and their values (include everything except the breadboard). Submit BOTH the “raw” electronic schematic file AND a PDF document suitable for printing. You may use any schematic capture tool you desire, but you are encouraged to use a tool which will allow ultimate import into PCB layout. You are encouraged to use Cadence/OrCAD or EagleCAD.

3. Screenshot or other documentation as supporting evidence of proper transfers at the proper bitrates

4. Code tree of your developed firmware (with all code adhering to the ECE 4723/6723 coding conventions)

5. Code tree of your developed PC library/objects (with all code adhering to the appropriate coding conventions from the community supporting your chosen language)

6. Code tree of your developed test benches

---

\(^3\)Submit electronically to lab TA.
6 Bonus

One of the benefits of using OO languages for embedded systems programming is the ability to encapsulate devices into classes that can be reused and which hide the details of the operation of the device. This bonus will provide 10 extra points if you encapsulate two distinctly different $I^2C$ devices.

Obtain a couple of distinct $I^2C$ devices from the TA, or procure your own favorite $I^2C$ devices. Encapsulate each device by creating an object that represents the device. The details of the actual $I^2C$ commands and transfers to operate the device are completely hidden from the application-level code. Your new objects must provide methods by which the application can access and use the device without need of understanding how the accesses are physically realized. In theory, the entire $I^2C$ command structure could change, or the devices could use some completely different communications interface without requiring a change to the application level code (apart from instantiation and a few initialization calls like setPortName and setAddress).

Your new object must present a complete interface to all functions described in the device’s datasheets. At least one of your devices must implement both $I^2C$ reads and writes. You must provide complete testbenches for both objects that demonstrates that your new objects function properly and are complete.

7 History

v1.0 5 SEPT 2006 – Initial version

---

4This bonus only applies to implementations in Java or Python