MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 7

Lab Guide

Department of Mechanical and Industrial Engineering

University of Toronto

Introduction

This lab focuses on sending data between two or more microcontrollers using a **communication protocol**. We will be implementing a **software version** of the standard **I2C** protocol, which we also cover in lecture. You will use this protocol to send short messages between your microcontroller and a neighbour. Using this functionality, we will then build a larger **network** of stations (each communicating using I2C) which can send messages around a **ring network configuration**. When complete, you will be able to send text messages between any two stations (of many) around the ring. Multiple groups will need to work together to complete this lab.

To begin, we will first revisit basic **digital input**, using the keypad as an example. We can later use the routines developed here to enter our text messages.

Keypad Interfacing

The keypad is arranged as a 4x4 array of switches. Pressing a button connects one of four **row pins** to one of four **column pins**. In order to determine which key(s) are pressed, we need to actively scan through the keypad.

On the evaluation board, the keypad occupies all pins on Port A. In order to determine which key has been pressed, we use Port A to actively scan through **columns.** The connections of the keypad are as follows:

PA0	PA1	PA2	PA3	
1	2	3	A	PA4
4	5	6	B	PA5
7	8	9	C	PA6
*	0	#	D	PA7

To determine which keys are pressed, we scan by first setting **only** PA3 to '1', then **only** PA2, and so on. This method energizes one column of buttons at a time. If a button is currently pressed, it forms an electrical connection between one of PA0..PA3 and one of PA4..PA7. When we energize each column using PA0..PA3, we can simply check which of the pins PA4...PA7 is equal to '1' to determine which buttons in that column are pressed.

This process is made flexible by the routine 'scanKeypad()'. This routine implements the scanning procedure by using an array of fixed **scan codes**, called 'scanCode[ ]'. Basically, these codes are successively outputted to Port A to determine the sequence of rows to check. We can modify this array to select which rows to test, and in what order. The routine also implements a mapping table called 'keypadTable[ ]' which links different physical button locations with different return codes for the routine. This can be used to control what character or value a physical key represents in your code. Note that the code as written only checks for the **first** key pressed; it assumes no multiple key-presses will occur.

You can make use of this code structure in your future projects as a simple method of collecting user input when your program is running. In combination with the LCD, we now have rudimentary User Interface (UI) tools, which make it much easier to interact with the projects we create – far less tedious than using only the debugger.

General Code Structure

Recall from the lecture on communications protocols and general input that certain types of input data and sources are best supported by different code structures. For this lab, we have chosen the simplest possible structure – a polling loop – which continuously checks for (i) user input, and (ii) incoming messages. It gives equal priority to both tasks, and does not implement any multi-tasking or interrupts.

Immediately, you should notice that this is an inefficient use of the microcontroller. Both tasks have relatively low throughput and high latency (time between events), meaning they would be much better suited for an interrupt-structured program. In your projects, you should try to be aware of these considerations. Since this particular program does not do anything else (it does not have any 'fast' components requiring constant updating, such as a central control loop), we can tolerate the inefficiency to allow easier debugging and better understanding as a teaching lab.

As we have established, the program implements a basic polling loop. Each time around the loop, we will do the following in order:

1. Scan the keypad for new key presses and store the result.
2. Check for and receive any new incoming message.
3. Re-send a message to next station if it is not for us.
4. Process any new key press (write to LCD/message buffer).
5. Send a message if we are done writing.
6. Repeat.

The process we outlined in the previous section for checking the keypad has been moved to a subroutine which is called to implement Step 1. It returns either a scan code representing the key currently being pressed or zero if no key is pressed. The previous key state is also stored to allow us to detect a complete down/up/down press cycle – in essence, this forms a simple bistable multivibrator or switch de-bounce method, like the one you implemented in last lab. The multivibrator state is first **set** when a key is pressed down, and then **reset** when the key is released. The reset triggers our program to actually process the key that was pressed.

State Machine Encoding

The second step of the procedure listed in the previous section requires us to use a new structure which we will study more of in the second last week of class – the **state machine**. In a state machine program structure, we maintain an explicit **state variable** for our program, representing the current *operating state* of the message-typing system. The state for this program can be represented as follows:

1. Basic – Typing a message, Typing recipient, or Sending/Idle
2. Typing Message or Recipient – Idle, Key Down, and Key Up (generates Key Press Event)
3. Key Press – Cycling through characters and numbers

To understand the concept of state, first think about a functional description of a system to record a message using the keypad. At the most basic level, we will be doing one of three things: (i) typing the message itself, (ii) typing the recipient's ID, or (iii) typing nothing / sending a message. We store the main state of the program in a **state variable** (see if you can identify it in the given code). This **explicitly** records what the program's state is for this aspect.

When typing either a message or recipient, we need to record and process key press events generated by the keypad. The earlier scanning technique can detect when a key is 'down' (i.e., connecting a row and column). However, a complete event requires this key to be released, as per our description of the bistable multivibrator concept. This is represented in the state machine program by a **hidden** or **implicit state** – we do not explicitly store this state in a variable; rather it is represented by the structure of the program itself. We use a series of if/else statements which, based on the typing mode, last key code, and current keypad scan result, determines when a complete key press (down then up) has occurred.

Lastly, in order to type complete messages with all 26 letters plus numbers, we need to detect when the same key has been pressed multiple times in a row. If this occurs, we will cycle between the letters associated with that key. This process uses an explicit variable to store what 'key index' we currently have. This state doubles as an ASCII offset to generate the correct character. The user interface of the program is as follows:

LCD OUTPUT

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
C1	M	:	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	R	:	🟥
C2	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩

The blue area denotes space for the message you are typing. The red area is the space for the recipient; with one hex digit, we can have a total of 16 'users' connected to our communication 'network' at a time. Lastly, the green area will be used to display incoming text notifications and status messages. Multiple functions have been added to 'advancedLCD.c/h' to support moving the cursor around the LCD. Please examine them now (in Lab7_1.mcp) to learn their use. In particular, note the function of moveLCDTo( ), printLCDText( ), and printLCDValue( ).

User Controls

The user will need to be able to enter plain-language messages to send to others, but we have only 16 keys to do so. We will also need control keys to switch between fields and for convenience functions, such as backspace, etc. The keypad layout will be as follows:

1 A B C	2 D E F	3 G H I	**A** End character
4 J K L	5 M N O	6 P Q R S	**B** Backspace
7 T U V	8 W X Y	9 Z	
* * + ,	0 0123456789	# Space	**D** Mode Select

Notice that most of the above keys share multiple characters; pressing the same key multiple times will **cycle through** the characters listed. Note the subtly different states required for each key:

Key '0' – Cycles through all numbers 0 through 9 when pressed

Key '2', '3', '4', '5', '6', '8', and '9' – Cycles through **three** letters (indicated under number)

Key '7' – Cycles through **four** letters (PQRS)

Key '*', '#' – Space and punctuation, three characters

Key 'A' – Ends the current character and starts a new one (use to generate repeated characters, for example 'TT' needs this key – pressing the '8' button will continue to cycle the first character between T-U-V until A is pressed.

Key 'B' – In message typing mode, this key acts as a backspace if you make a mistake.

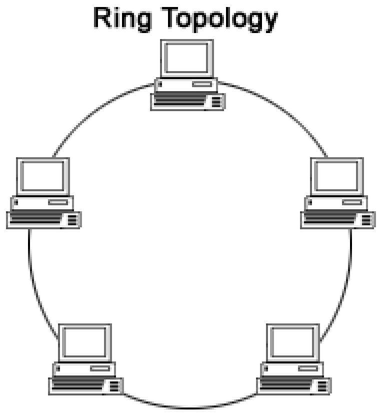
Key 'D' – The next/send key; in message-typing mode this ends typing and moves to recipient typing, in recipient typing this **sends the message** and returns to message typing.

To use the program, simply type your message first, then press 'D'. The recipient must be correct (as we will see), otherwise communication problems will occur. In particular, recipient is represented by a single HEX character (0-9, A-F). Notice that there is a #define for **MACHINE_ID**. This sets the code for your station – change it to something unique (ask other groups). If it is not unique, other groups may intercept your messages.

Now that we have the basics of the keypad system, let us look at the communication protocol and structure that we will be using.

General Communication Method

For this lab, we will be using what is called a **ring topology** for message passing (special thanks to Osmond Sargeant for suggesting this). In a ring topology, each station has two lines, one incoming and the other outgoing. We connect stations together output-input to form a ring:



Note that the individual communication protocol used can easily be adapted to one-directional communication between just two boards. In fact, the simplest ring consists of just two stations. To send a message **to anyone on the loop**, we simply send a **broadcast** containing the message and a **header** with the recipient. This message is always sent first to the nearest neighbour connected to *our* **RING_OUT** port (the output of our station). The next station examines the message, and if it decides it is the recipient, it keeps the message and the send is complete. Otherwise, it *passes* the message to the next station, and so on around the loop. This manner of communication was the standard for early wired networks (search **Token Ring** for a typical example) before the Ethernet (IEEE 802.3) standard with hubs/routers became popular. It is still used in simple distributed computing networks, due to the low overhead needed.

This scheme has a number of interesting details that we should be aware of:

(i) If the loop is long, it can be inherently somewhat wasteful. A message will often have to be passed along by multiple stations before reaching its destination. If many messages are being passed, you may see noticeable slow-down of the network or even the individual station. If there are n stations, the message (worst-case) may have to pass through $(n-1)$ stations to reach its destination.

(ii) For large-scale applications, data security is an issue, as your message can easily be intercepted by other (non-recipient) stations. This is often called a **man-in-the-middle attack** in security; here the vulnerability is essentially built into the design. For local or non-secure communications this is not an issue.

(iii) The **recipient must exist** in the loop. If not, the message has the potential to continue around the loop infinitely. This is a definite possibility in the lab. The solution is to include the source station in the header sent with the message. Each station can then check messages it passes, and if it is the originator, it can drop the message – in this case, the message has passed around the entire loop without being received.

(iv) Recipient addresses **must be unique**, otherwise accidental interception may occur.

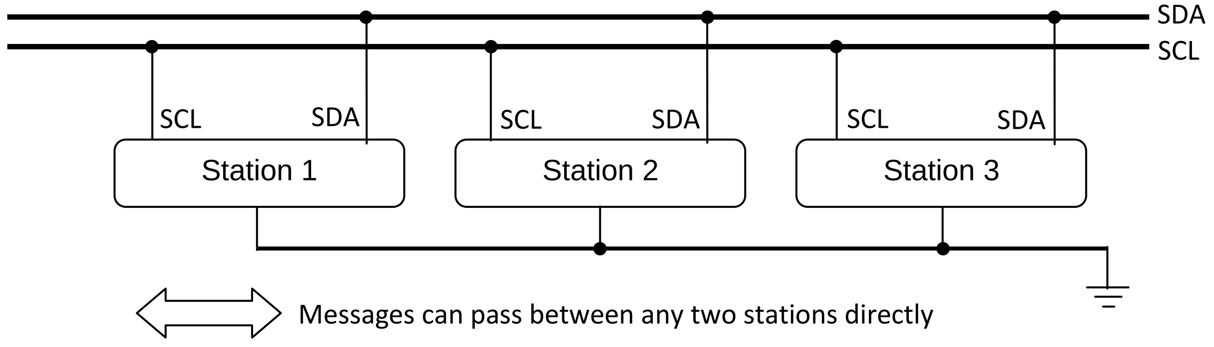
This covers the basics of the high-level structure we will be setting up. Now, we need to work out the details of low-level communication.

Low-Level Communication Structure – The I2C Protocol

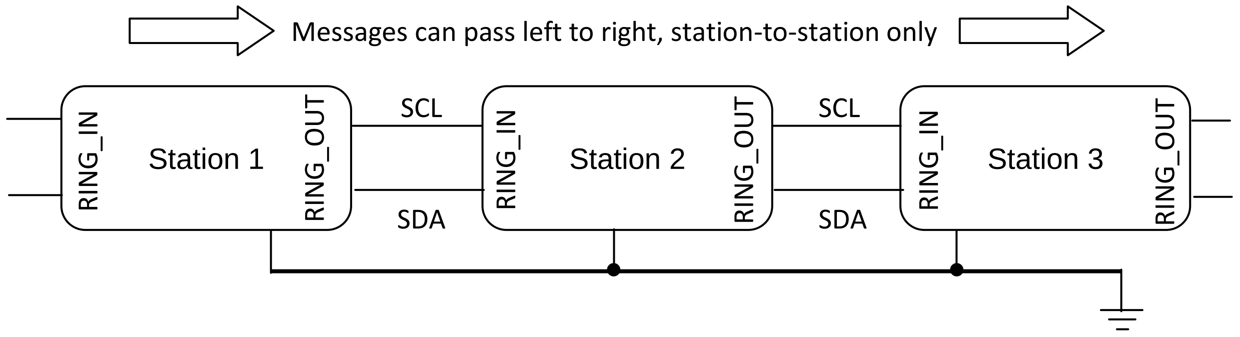
For this lab, we will be using a slightly modified **software** implementation of the I2C protocol. This protocol was originally developed by Philips Semiconductor (now NXP), and is a common standard used by various sensors and peripherals for communication with other hardware, such as a microcontroller. In fact, many modern microcontrollers have hardware-level support for I2C built-in. However, the message format and handshaking sequence which define I2C can be implemented on any microcontroller in software, provided that we have two or three general-purpose input/output pins available.

The protocol itself is a **serial standard**, meaning that data is sent bit-by-bit over a single wire, the data line (labelled **SDA** for this standard). The lowest level of synchronization is achieve using a second wire which provides a clock signal (labelled **SCL**). Each pulse on the clock line matches and synchronizes the transmission of one bit of data on the data line.

I2C also allows for more than two devices to be connected on shared SDA and SCL lines. If we desired to, we could build a network of interconnected stations using only one I2C bus connecting multiple stations:



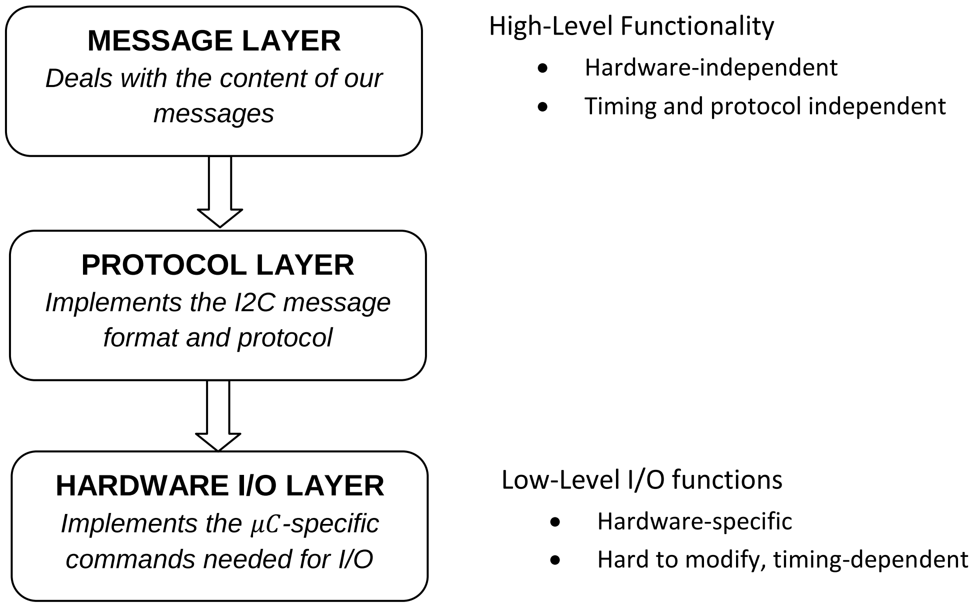
However, there are limitations with the above arrangement. I2C is already a relatively slow communication protocol. The original specification (Version 1) calls for communication at only $400\,KHz$. Later variants have increased the speed up to $5\,MHz$ (with the latest ultrafast mode, UFm). But, not all hardware can support such speeds. There are issues with driving multiple devices attached to the bus at once, and noise pickup is an issue. To avoid these issues, we will use a chain of individual I2C busses to implement our ring network:



Each microcontroller station will implement **two** I2C networks; one called **RING_IN** connecting our station to a **sending station** to our left (where our microcontroller is a **bus slave**), and one called **RING_OUT** connecting to a receiving station on our right (where our microcontroller is the **bus master**). In network terminology, the bus master controls the operation of the bus; it is the station that decides who can use the bus to send/receive data and when. Only one device can use the bus at a time, so the bus master must **arbitrate** to avoid conflicts. Technically, the I2C standard is a **multi-master, multi-slave** standard. For our purposes, we are only implementing unidirectional transfer of messages, so each I2C network can be greatly simplified.

Communication Layers

To better organize and abstract the work at hand, we will divide our implementation of the I2C protocol into **layers**. Each layer collects together functions of a similar level of abstraction; low-level hardware-layer functions are generally microcontroller-specific and deal with setting up the actual hardware input/output pins, sending 1's and 0's over these pins, and other similar low-level functions. At the opposite end, high-level functions define the message format the protocol expects. The high-level functions call low-level functions to actually send and receive such messages, but do not really depend on the details of how the low-level I/O is accomplished. This **abstracts** the low-level functions away from the message format, making it easier to implement and modify – it is good coding practice. The three layers used in our implementation are shown on the following page.


MESSAGE LAYER
Deals with the content of our messages
PROTOCOL LAYER
Implements the I2C message format and protocol
HARDWARE I/O LAYER
Implements the μC-specific commands needed for I/O
High-Level Functionality
• Hardware-independent
• Timing and protocol independent
Low-Level I/O functions
• Hardware-specific
• Hard to modify, timing-dependent

1. The **Message Layer** implements the **content** of the messages we want to send. For instance, our messages are planned to be a string of ASCII characters, up to 10 bytes in length. Our message will also contain a recipient address, and a simple error check in the form of a 'length of message' field. This layer is completely hardware and protocol independent. We will write the function only in terms of generic functions – sendChar( ), and recieveChar( ), for instance. Assuming these can be implemented, the details do not matter to this high-level function. The use of I2C could be substituted with an entirely different protocol without changing the code of this function. No timing-specific or hardware-specific code would be included for this reason.

2. The **Protocol Layer** implements the **format** of messages which is specific to the chosen protocol, I2C. It may implement timing and other specific features of the protocol specification, but will do so using functions which are still generic across microcontrollers – for example, 'setSDA( )' or 'I2CStart( )'. In this way, the protocol functions can be implemented independent of the details of the microcontroller's hardware – we could change the 68HCS12 board for another microcontroller without changing this code.

3. Finally, the **Hardware I/O Layer** contains all of the microcontroller-specific code required to set up digital I/O over the SDA and SCL lines. For instance, we may set SFRs to make certain pins inputs and outputs, or implement an interrupt system. These functions are specific to the microcontroller we are using, but can be replaced when moving to a new platform without changing **any** of the functions in the layers above. It is this **abstraction** of function that allows for simple code **and** hardware modifiability – it is good programming practice to follow.

Hardware I/O Layer

In this lab, we will be using a total of **six** digital input/output pins on each microcontroller to implement **two** I2C busses per station. The assignment of pins is as follows:

RING_IN – Accepts data **in** from the station to our left. At our station, Port B is used: PB0 is SCL and PB1 is SDA. We also add a separate line for acknowledgement, called ACK, using PB2.

RING_OUT – Sends data **to** the station on our right only. At our station, Port T is used: PT0 is SCL, PT1 is SDA, and PT2 is ACK.

The total code for the hardware I/O layer will consist of the following:

1. A section at the start of main( ) which makes PB0, PB1, and PT2 **inputs**, and PB2, PT0, and PT1 **outputs**. This allows basic digital I/O over these pins.
2. A series of defines which replace the specific pin assignment (PB0, PB1, etc.) with generic names (RING_OUT_SCL, RING_IN_SDA, etc.). This maintains the separation of hardware-specific code and the later protocol-layer functions.

For other microcontrollers, the series of defines might be replaced by macros or even functions. This might typically be the case if sending a '1' or '0' at the lowest level requires something beyond code similar to 'X = 1'. For instance, if we need a delay added **per-transition** (i.e., 'X=1; … delay(n); … X = 0'), using a macro function in the hardware layer would be the best place to implement it.

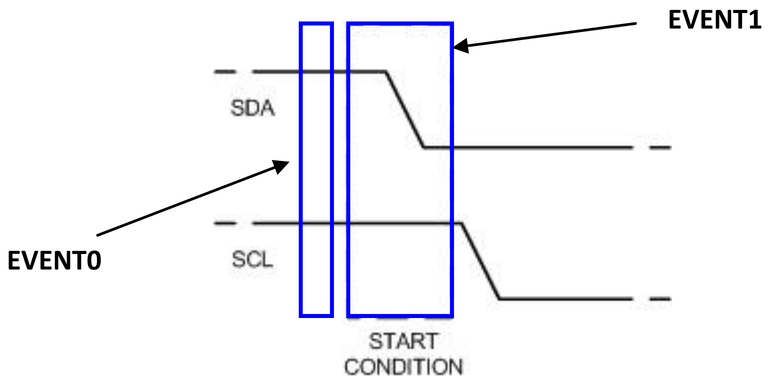
Protocol Layer – I2C Details

The protocol layer for this lab will implement a variant of the generic I2C protocol. This protocol defines a two-wire or three-wire serial message format. To analyze this format, we will divide it into states or **condition** under the I2C terminology. Consider the idle state, where the I2C bus is doing nothing:

IDLE CONDITION: SDA = 1 (master), SCL = 1 (master), ACK = 1 (slave)

In the idle state, all wires are pulled to a logic '1' level. Depending on the hardware, this will usually be represented $+V_{DD}$ appearing at the corresponding output pin. For the 68HCS12 variant in the lab, this will be $+5V$. Note also that the bus master is responsible for **asserting** (setting) the SDA and SCL lines to '1', while the slave is responsible for ACK. Since we are implementing only one direction of communication (master-to-slave), this assignment will remain the same at all times and will be omitted from the explanations that follow. The full I2C specification will have different assignments in different states, in order to allow bidirectional transfer.

When we want to begin an I2C transmission to the slave device, the bus master creates the I2C **START CONDITION**:

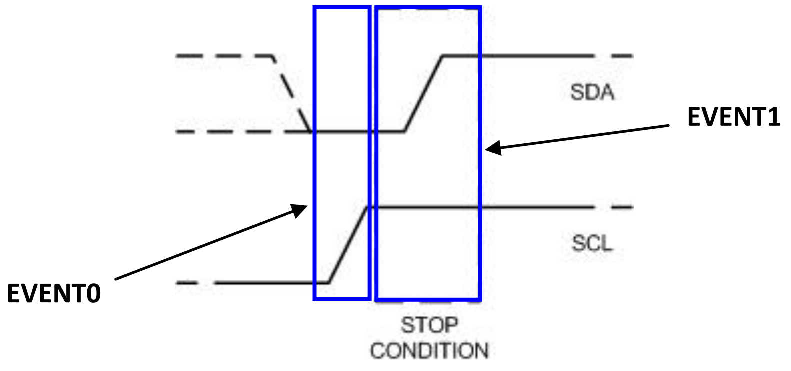


In the above **logic transition diagram**, we represent each line as either low or high. Notice that we can break the I2C start condition into two key events:

EVENT0 – A precondition; the SDA and SCL lines should remain a stable '1' for a period of time before the start condition occurs.

EVENT1 – The start condition itself; SDA transitions 1→0 while SCL is held stable 1→1.

A full message (sent or received) under this protocol is **framed** by both a start condition **and** a **STOP CONDITION**.

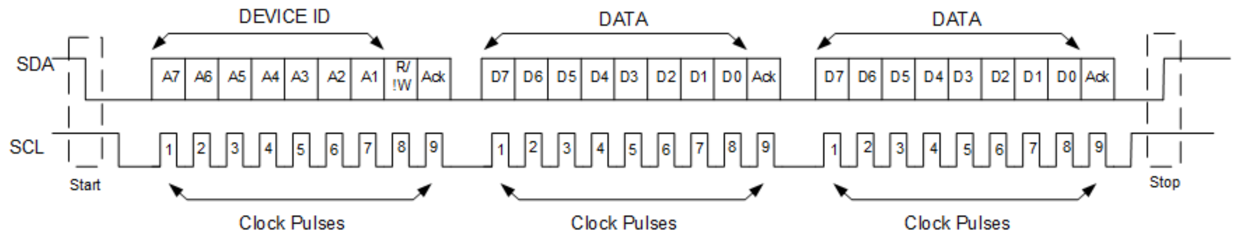


Like the start condition, there are two key events that we can identify:

EVENT0 – The SCL line is returned from 0→1 **before** the stop condition is generated. SDA should be a stable 0→0 during this time.

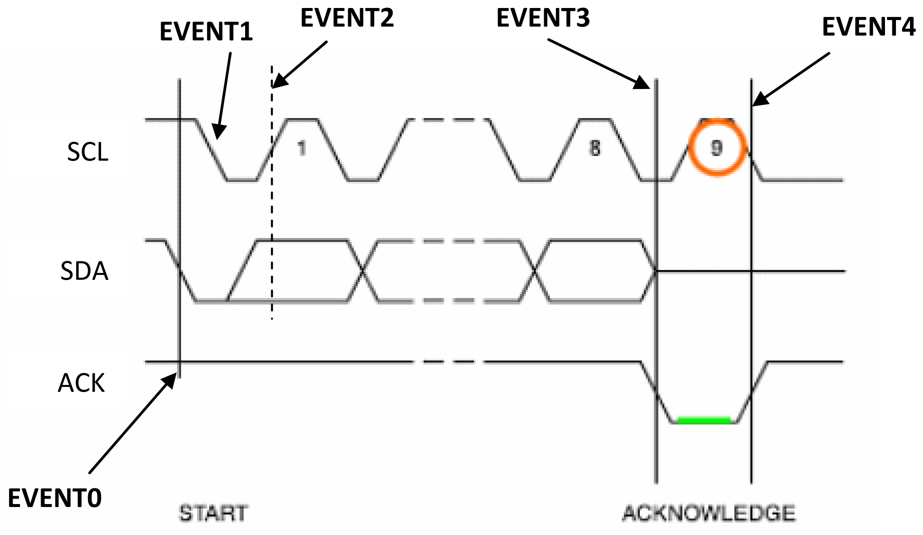
EVENT1 – The stop condition itself is defined as a transition of SDA 0→1 while SCL is held at a stable 1→1 for a certain duration.

Given the above definition of the start and stop conditions, a complete I2C message can be framed using the start and stop conditions as endpoints:



The above is an **example** message, containing two payload data bytes. The I2C protocol specifics **only** that the first byte of any message sent should be a DEVICE_ID byte; the remaining bytes are payload, and may be of any desired total length. Within the DEVICE_ID byte, the **slave address** is included. Recall that the I2C standard allows multiple slave (receiving) devices to connect to the same bus. All devices listen, and the one with an in-built slave address matching the bits in DEVICE_ID will take precedence as the device to receive data. This practice introduces a practical upper limit to the number of devices on the bus. The DEVICE_ID byte also contains a **R**ead/**W**rite bit, as the protocol does allow bidirectional data transfer. Since our implementation transmits data in one direction only, and will only ever have one device connected, **we will completely omit this byte** from our implementation.

All that remains is to specify the timing of the data byte transmission itself. From the above diagram, you may notice that each byte actually requires **nine** bits to be sent on the bus. The first eight bits are the data sent by the master, and the ninth bit is an acknowledgement of the received data, sent by the slave device. The I2C specification allows this to be sent over the SDA line if desires. However, to make our implementation simpler we **add** an extra wire – the ACK line from earlier. In this way, the SDA line is always an output at the bus master, and the ACK line is always an input:



We can now define the events of a data **byte** transmission:

EVENT0 – Our precondition for the receipt of a data byte is **either** a Start Condition has just been generated **or** we have just finished a prior byte (ACK has already been sent). The bus state should be SCL = 1 and SDA = 0.

EVENT1 – At the start of the first clock cycle, we have SCL 1→0. The signifies the start of the time period where the sender may change SDA.

EVENT2 – SCL is held low (SCL = 0) for a defined amount of time. This is the amount of time available for the sender to correct set SDA to the data bit being sent. At the end, SCL transitions low-to-high (SCL = 0→1), and data is captured on this rising edge by the receiver (slave device). The data on SDA should remain stable (unchanging) for a minimum amount of time to ensure proper capture.

The above events are repeated eight times, starting with the MSB of the data being sent first, until the complete byte is sent.

EVENT3 – After 8 bits are sent, there is a final transition, high-to-low (SCL = 1→0), indicating the end of clock pulse 8. At this point, the sender (bus master) **waits** and does nothing further. The receiver (slave device) is responsible for asserting the ACK line low (ACK = 0) within a limited time.

EVENT4 – Once the bus master sees that ACK is asserted zero (ACK = 0), it delays a short time, then asserts SCL 0→1, delays again, and asserts SCL 1→0. This generates the ninth and final clock pulse on SCL. The falling edge of this pulse (SCL 1→0) triggers the receiving slave device to de-assert ACK (now ACK = 1). The sender also returns the SCL and SDA lines to default values. The bus is now ready to either send another byte **or** the stop condition.

This completes the description of the I2C protocol. We are now able to send arbitrary messages to a receiving slave device. The final step will be to decide on the content of said messages in the Message Layer.

Message Layer

As identified in our earlier design of the software, the message layer is independent of the communication medium and its implementation. It would not matter if we use I2C, SPI, or another protocol – so long as we can send an arbitrary payload of *n* bytes, this message format can be used. In this way, we are practicing good coding methods by using abstraction.

For the task at hand, we wish to send a message of characters to another station. We need to send to the receiving station (i) the message itself, (ii) the length in characters (since it may be of varying length), and (iii) an identifying character to select the recipient. It makes sense to transmit the length first, but others the ordering is entirely up to us:

LENGTH	**RECIPIENT**	**MESSAGE**
1 byte	*1 byte*	*[LENGTH] bytes*

In the above, we will enforce a limit on message length of 10 characters due to the size of the displaying LCD screen. This will be limited at the sender's side, but the receiver will also check the received 'length' value – this is good practice when using numbers from unsecured data sources. We are now ready to begin the lab and investigate the I2C protocol.

Lab Guide – Ring Communications

We are now going to guide you through the process of setting up your station locally first, and then joining a ring with your classmates.

Step 1 – Find a partner group (the closest group would be ideal). You will initially form a ring of two to test basically functionality.

Step 2 – Open the base code project, 'Lab7_1.mcp'. This code is ready to use in terms of communication, keypad input, and basic communication described above.

Step 3 – Change the #define MACHINE_ID to a unique value, compile the code, and download to the microcontroller. Run the program to ensure that everything works. Try keying in a simple message and recipient, then pressing the D key. This will initiate a send; you should notice the LEDs for Port B change. Turn off the power to your board and assist your partner group with this step if they are not yet done.

Step 4 – Once you have both tested basic functionality, form a ring with your two stations.

Connect your RING_OUT (pins PB0..PB2) to their RING_IN connection (pins PT0..PT2). The connections should be made as per the diagram on the following page, and should be made **with the power off**. Note the wire colors as indicated by the arrows. The ground will also have to be connected between each board. Have the TA check your connections before powering back up to avoid permanent damage to the microcontroller boards. It is best to use the GND terminal near Port B and Port T on the respective stations for this connection.

You			Them	
RING_OUT_SCL	**PB0**	→	**PT0**	RING_IN_SCL
RING_OUT_SDA	**PB1**	→	**PT1**	RING_IN_SDA
RING_OUT_ACK	**PB2**	→	**PT2**	RING_IN_ACK
Ground	**GND**	→	**GND**	Ground

All boards on the ring **must** share a common ground, otherwise the electrical signal levels at the output pins may not match, prevent communication.

Step 5 – Power both boards and take turns to send messages. Confirm that everything is working (**both sending and receiving**) before proceeding any further.

Step 6 – Try sending a message with an incorrect recipient ID. Note what happens in the answer to **Question 1** in the deliverable.

Step 7 – If no other groups are ready, you can assist other groups or start examining the code to answer the later hand-in questions. Once a second pair of groups is free, come together and begin to form a larger four-board loop. You can ask the TA for assistance with setting up this loop. Once it is running, try sending some messages to different recipients on the loop. The eventual goal is to connect **all groups** in the lab together.

Step 8 – Once most or all groups are connected to the loop, it is time to try a group experiment. Have one person send a message to an invalid recipient and record the result in **Question 2**. Continue sending more messages of this type until you reach a limit.

Step 9 – As a group with the class, make a modification to the program to resolve the issue with incorrect recipients. Attach a code listing for your modification for **Question 3**.

Step 10 – Continue to experiment with the larger loop (or in smaller loops) to gain insight into **Questions 4 and 5**. However, these can be answered outside the lab if you desire – they are longer, and are more designed to help you with exam-type practice.

**** End of Lab 7 ****

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 7

Deliverable

Department of Mechanical and Industrial Engineering

University of Toronto

Last Name, First Name	Student Number

Q1. What happens when you send a message with the wrong recipient? Are you able to do anything at all? If you are able to, what happens if you send a second message?

Q2. What happens when you send one or a few messages with incorrect recipients in the larger loop? How is this different from what happened in Q1?

Q3. Code listing of modifications to program.

Q4. Show how the program handles **keypad** input to type a message by drawing a diagram that represents the program flow (exclude communication tasks). You do not need to follow any particular format, but you should try to show what you think are distinct program states and the transitions between them (including what causes these transitions). We will learn more about this process in class when we formally study state machines.

Q5. Draw a timing diagram showing the three lines (SDA, SCL, and ACK) for a complete message being sent. The message should be short (one or two bytes).

Use separate pages for Q4 and Q5; they will probably take about one page each.