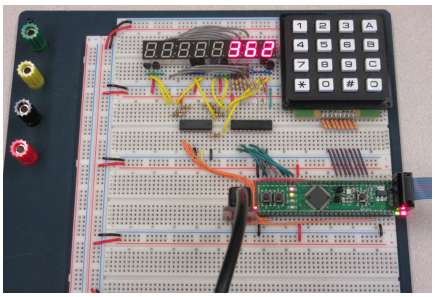
ECE 362 Lab Experiment 5: Timers

Home About References Lecture Homework Labs

Introduction



Each timer subsystem allows the microcontroller to produce periodic interrupts while the CPU is busy with other operations. In this experiment, we will use timer interrupts to periodically scan a matrix keypad and reliably report button presses. By recording the history of button presses with an interrupt handler, the main program is free to do other things, or it can wait for a button press to take place.

Instructional Objectives

- Learn how to configure the timer subsystem
- Use timer interrupts
- Reliably debounce a *matrix* of buttons
- Learn how to drive a *multiplexed* display

Table of Contents

2	Experiment	
2.1	enable_ports	5
2.2	Basic Interrupt Service Routine	5
2.3	Configure a basic timer	5
2.4	A display multiplexing and key scanning application	
2.4.1	Data structures for the application	
2.4.2	show_char: Display a single character	10
The work, above, must be demonstrated in lab to get credit on remaining parts.		
2.4.3	drive_column: Update the keypad column being scanned	5
2.4.4	read_rows: Read the row values	5
2.4.5	update_history: Update the history bytes for one column	10
2.4.6	Timer 7 ISR	10
2.4.7	setup_tim7: Configure timer 7	10
2.4.8	wait_for: Wait for a single keypad button press	10
2.4.9	shift_display: Shift all display entries left	10
3	Observations and thoughts	
3.1	Unrelated interrupts	
4	Submit your postlab results	*
	Total:	100

* All the points for this lab depend on proper completion of and submission of your post-lab results.

When you are ready for your lab evaluation, review this checklist.

Step 0: Prelab Exercises:

Do them in this order...

1. Be familiar with lectures up to and including Debouncing and Multiplexing.
2. Chapter 15 of your textbook is the most helpful thing you can read to understand the timer subsystem.

3. Read though Chapter 20 of the STM32F0 Family Reference Manual to become familiar with the basic timer (TIM6/TIM7) subsystem.
4. Read the datasheets for the hardware you will use:
   - The TDCR1050M 4-digit common-anode 7-segment display
   - The TLC59211 sink driver
   - The 74HC138 3-to-8 decoder with active-low outputs
5. Read this entire lab document.
6. Ensure that your development board and serial port are working **before** you start the lab.
7. After doing the previous steps, including reading the entire lab document, then do the prelab exercises and submit them **before** attempting the lab experiment.
8. Wire the keypad and 7-segment displays as described in section 1.5 of this lab document.

Step 1: Background and wiring

1.1 Debouncing buttons

Because switches are mechanical devices, they bounce. This means we do not get a stable electrical signal on a press or a release. Because of switch bounce, the voltage fluctuates between V_{DD} and ground before settling down to a stable value. There are several ways to fix this. Debouncing can be performed in hardware or software.

1) Hardware debouncing: You have already seen this working well in previous labs (lab 3). Hardware debouncing is done via a simple RC network to filter high frequency glitches that occur from mechanical bounce. The Schmitt trigger inputs of the microcontroller can tolerate the relatively low speed of the signal changes.

2) Software debouncing: In certain applications it might be beneficial to debounce switches in software rather than use an RC filter on each switch.

Software debouncing can be achieved in a variety of methods. The simplest debouncing method is to read the voltage, wait for the switch

to settle and read it again. However there are more robust approaches, which are more suited for a keypad matrix. Here we describe one such method that we dub the "history method".

In this "history" method we store N previous button states. Each button's history is represented by a single variable. This variable is referred from here on, as the 'history variable'. The least significant bit of the history variable represents the most recent button state and the most significant bit represents the oldest button state in history. When a new button state is read, the history variable is shifted left by one, making space for the new state in the least significant bit. The new state value (i.e. 0 or 1) is ORed into the history variable.

Now that we have a history of button states we can look for two specific patterns to identify a button press and release. An 8-bit pattern of **`00000001`** represents the first moment of a button press and **`11111110`** represents the first moment that of a button release. Note that these patterns are applicable when the button's logic is active high, else the patterns are switched. Therefore, by checking if the history variable equals 1 (i.e. 00000001) we can detect a button press. Similarly by checking if history variable is 0xfe (i.e. 11111110) we can detect a button release.

1.2 Reading a button matrix

A matrix of buttons presents an additional complication to the challenge of debouncing since it is not possible to monitor each button separately. To understand why this is so, it's helpful to discuss how a matrix keypad works.

You will use a 16-button keypad for this lab experiment. One configuration for such an arrangement of buttons would be to have two wires for each button (32 wires total) so that each button could be independently monitored. That would require a lot of pins on the keypad which might make it difficult to use on a breadboard. An optimization might be to have one *common* power connector and one more connector per button (17 wires total) so that each button could still be monitored individually. But this is still too many pins on the

keypad than manufacturers will usually support, and the numbers would only be much worse for keypads with even more buttons.

The typical concession for keypads is to have multiple *common* pins that connect to groups of buttons. The logical extension to this trend is to configure a keypad as a *matrix* of rows and columns. If the number of rows and columns is about equal, this practice minimizes the number of pins on the keypad. The added challenge is that the keypad must be scanned one row or column at a time. For instance, a voltage can be applied to a column of buttons, and the rows can be sensed to detect if any of the buttons in the column are being pressed. Then the voltage can be removed from the column and re-applied to another column. By cycling through all the columns, all keys can eventually be detected. The difficulty is that, not only do the buttons still bounce, the bouncing must be tracked over multiple successive scans.

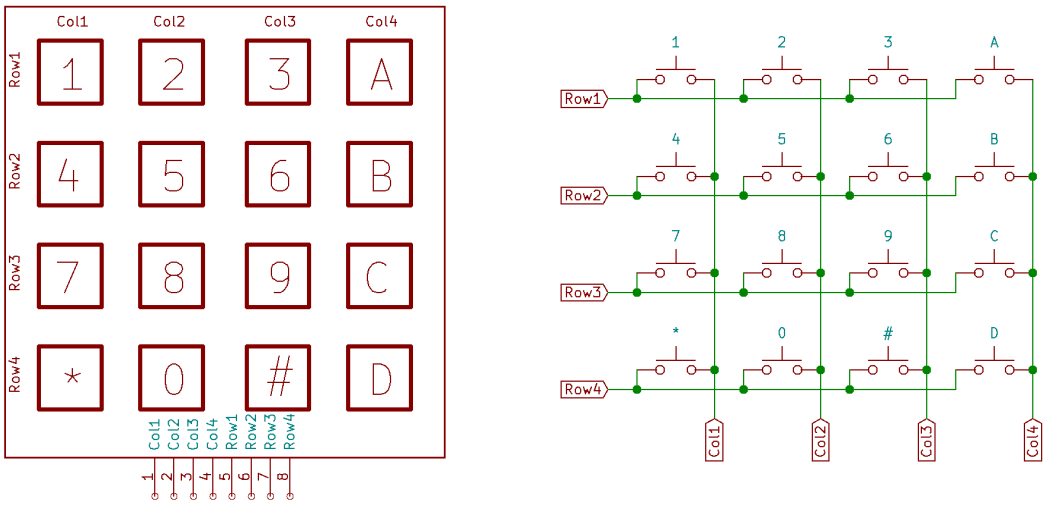


Figure 1: Keypad schematic (click for larger view)

The schematic for the keypad matrix you will use for this lab experiment is shown in Figure 1. For this experiment, you will implement the 'history' method of debouncing buttons. For each of the 16 keys, an 8-bit variable is used. Each history variable is stored as an element of a 16-byte array in RAM. The array's first 4 elements, 0 – 3 represent the history for the first column of keys, the elements from 4 – 7 represent the history values of the second column of keys and so on. Since buttons will be sampled four at a time, the history values will also be updated four at a time. This is effectively equivalent to sampling a

single button at a time except that the samples are further apart. Button presses and releases are still detected the same way.

Another minor downside of this method is that it is not always possible to detect multiple simultaneous button presses. For instance, if Col1 has an applied voltage, it is possible to detect the '1' and '4' buttons pressed simultaneously because they will be connected to distinct rows. It is also possible to detect two buttons in different rows and different columns. Nevertheless, if two buttons in the same row are pressed simultaneously, the results will be unreliable. For instance, if '1' and '2' are pressed, when Col1 has a high voltage applied, Col2 will have a low voltage applied, and vice-versa. The value read on the Row1 will be indefinite. There is a way to overcome this difficulty: use pull-up resistors on the columns as well as the rows, and configure the driven pins to use open-drain outputs so that they can only be pulled low. Then, the presence of a high voltage read on the row pins always indicates "no button in this row pressed" and a low voltage on one or more row pins indicates a button press. (Inverted thinking, like this, is difficult when programming in assembly language. We might give this a try in the future, but not now.)

In the event that logic high and logic low are applied in a way that current flows through a button, the keypad button may be damaged. In this experiment, you will be driving every column either high or low. If two buttons are pressed in the same row, that could mean be lots of current flowing from one column to another. We don't know the current rating of the keypad switches, but we know they weren't made for this. For this reason, you will use 1KΩ series resistors between the STM32 and all of the pins of the keypad. By doing so, there cannot possibly be any damage.

1.3 Driving multiplexed displays

A 7-segment display is one that has seven LED segments in the shape of a number "8". By selectively turning segments on, other numbers can be displayed. In practice, such displays have more than seven segments. An eighth segment shows a decimal point. Recall Figure 2, from ECE 270, that shows the names of the segments of a display. The top segment is customarily called "A". Proceeding clockwise, the next ones are named "B", "C", and so on. The middle segment is named "G". The decimal point is named "DP".

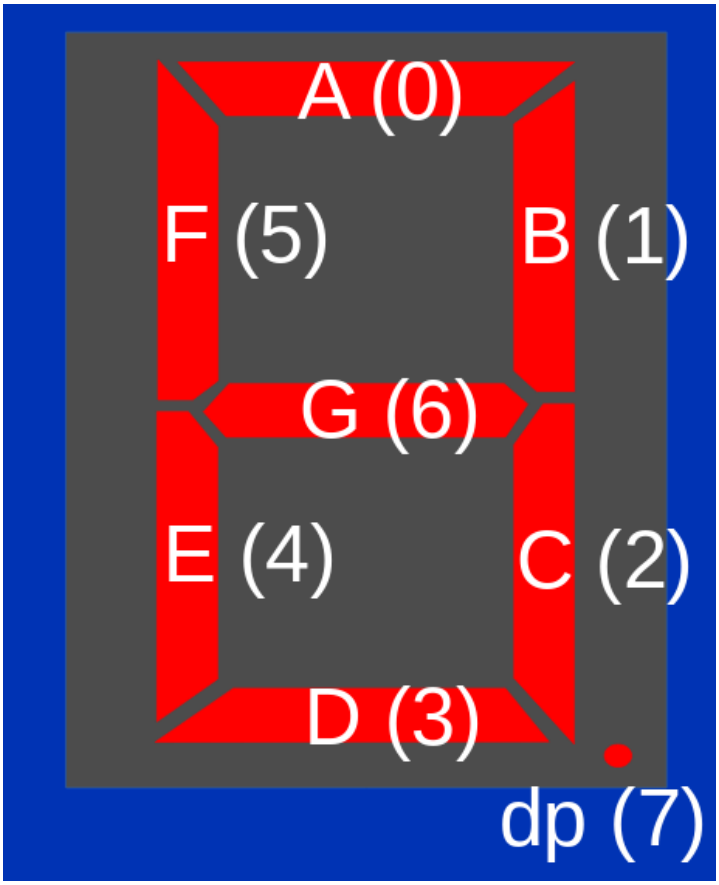


Figure 2: Seven-segment display organization

For this lab experiment, you will be using the TDCR1050M 4-digit 7-segment LED display. This device was made to be used in a clock, so it has a middle colon for use in displaying a time. Another dot on the display can be illuminated to indicate things like an alarm setting. For this experiment, we will use only the digits and decimal points.

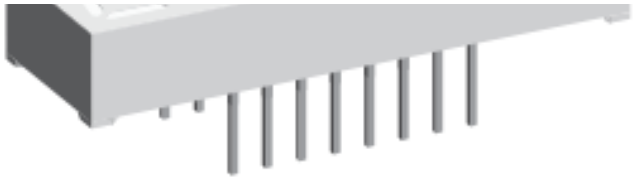


Figure 3: Picture of physical display

If you look at a picture of the physical device in Figure 3, you see that it has only two rows of eight pins on each side. Only sixteen pins are used to control more than 30 individual LED segments that must be independently lit to form four distinct digits and decimal points. To do so, it uses a "common anode" configuration so that one pin is connected to the anodes of all the LEDs on one digit. There are four distinct pins connected to the anodes of the four digits. There are another eight pins connected to similar cathodes on all the displays. For instance, one pin is connected to the four "A" cathodes on the four digits. This means it is possible to illuminate all four of the "A" segments on the four digits by applying a positive voltage to the four common anodes and a negative voltage to the pin connected to all of the "A" segments. It is not possible to simultaneously illuminate the "A" segment of all the displays and illuminate the "B" segment of only a single display. Figure 4 shows the schematic for the connections to all of the LED segments for the display.

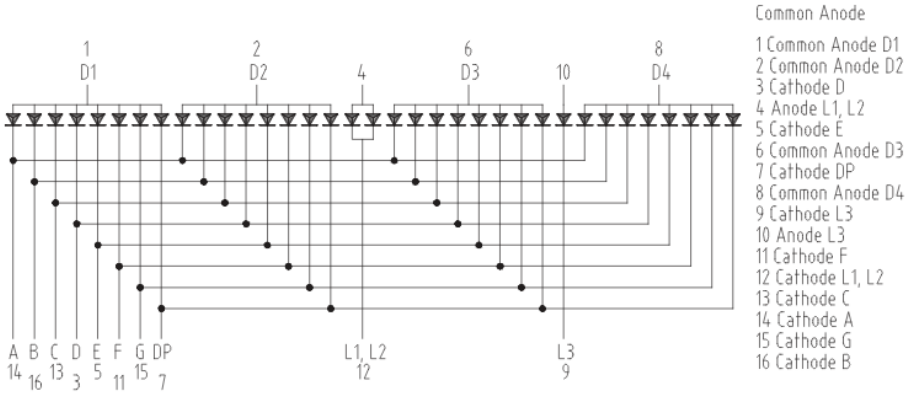


Figure 4: Schematic for the TDCR1050M display

If it is not possible to independently turn on and off individual segments at the same time, how is it possible to display distinct numerals on the four groups of segments? This display is made to be *multiplexed*. In a manner similar to how we cannot detect the exact moment that a

button in a matrix is pressed, we cannot simultaneously turn on every LED segment we need. Instead, the segments of the first digit are turned on, then the segments of the next digit are turned on, then the third, then the fourth. If each individual digit is illuminated for one second, then the display will not be perceived as a four-digit number. If, however, the cycling through the digits is done quickly enough, it will *appear* that all of the numbers are shown simultaneously. This is because of the human eye's *persistence of vision*. A flickering light—as long as the flickering is rapid enough—is perceived as continually on, although maybe dimmer than normal.

As an example, consider the result of applying the following cycle of voltages applied to the pins of the display:

- A positive voltage to pin 1 (D1), and a negative voltage to pins 16 (B) and 13 (C)
- A positive voltage to pin 2 (D2), and a negative voltage to pins 14 (A), 16 (B), 3 (D), 5 (E), and 15 (G)
- A positive voltage to pin 6 (D3), and a negative voltage to pins 14 (A), 16 (B), 13 (C), 3 (D), and 15 (G)
- A positive voltage to pin 8 (D4), and a negative voltage to pins 16 (B), 13 (C), 11 (F), and 15 (G).

When these four configurations are applied repeatedly and at high speed, it will *appear* that the four-digit number "1234" is present on the display.

How rapidly must you cycle through each digit to achieve the effect of persistence of vision? Usually, anything flickering faster than 30 times per second appears to be continually on. As long as all of the digits of the display are turned on and off more than 30 times per second (even though none of them are on at the same time as others) the display will appear continuous. In this lab experiment, you will use two groups of the four-digit displays (eight digits in total), and you will turn on each digit for one thousandth of a second. By illuminating each individual digit for one-thousandth of a second, any one of the eight displays will be illuminated eight times less frequently (or 1/125th of a second). This is much faster than is needed to perceive the digits as all being on simultaneously.

1.4 Use of a sink driver and high-side transistors

Most students, when they approach the idea of multiplexing, would be more comfortable with a "common cathode" display instead since it could be configured so that a positive voltage applied to the pin connected to the anodes of all "A" segments would illuminate that segment when a negative voltage is applied to the pin connected to all the cathodes of the digit. If you were connecting the pins directly to your microcontroller, that might even make sense. Nevertheless, the current-flow requirements for the LEDs make it impractical to illuminate many of them using only the capabilities of the microcontroller. Instead, it is common use *sink driver* connected to the cathodes of each type of segment (e.g., "A", "B", "C", etc...). A sink driver is a device that is capable of sinking (connecting to ground) a connection with a large current flow. 47Ω limiting resistors will be placed in between the sink driver and each cathode to prevent too much current from flowing through each segment. Conveniently, the sink driver will sink current through an output pin when its corresponding input pin is high. In this way, it acts as an open-drain inverter. (A sink driver cannot push an output pin high.) Using the sink driver, a logic high applied to the driver will cause the particular segment to illuminate, which is how you would like to think about it.

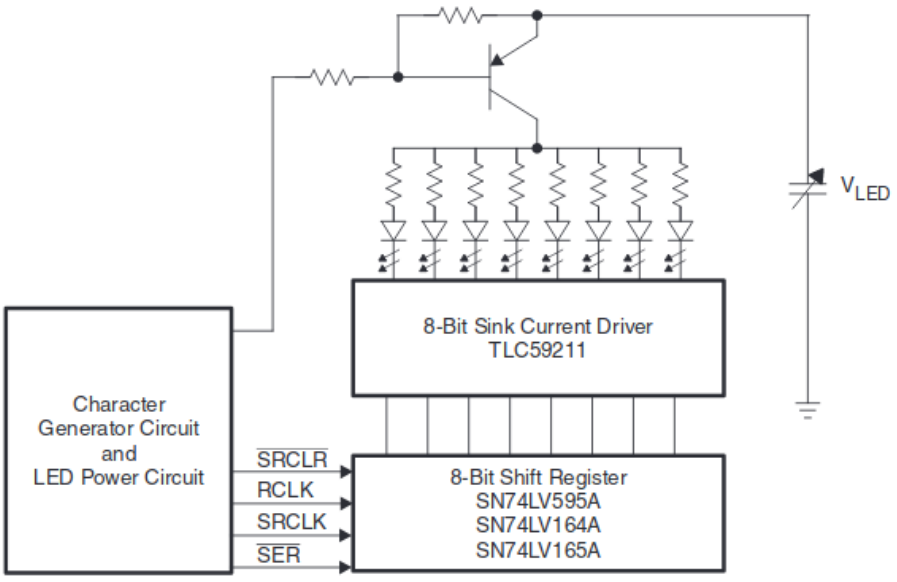
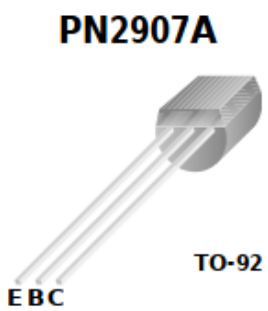


Figure 5: Typical LED display circuit

Consider the circuit in Figure 5 that appears in the TLC59211 datasheet. In this lab experiment, you will wire the microcontroller to the sink driver rather than an 8-bit shift register. The device at the top of the diagram is a PNP bipolar junction transistor (BJT). This device will conduct current through the top pin (the emitter) in the direction of the arrow, and out the bottom pin (the collector) when the middle left pin (the base) has a voltage lower than the emitter (and a small current coming out of it). It is conceptually similar to a P-Channel MOSFET except that a MOSFET does not require a substantial current flow through its gate. By contrast, the collector current for a BJT is directly proportional to its base current.



You will use one PN2907A PNP BJT (in a TO-92 package) for each anode of each digit of the display. Pay careful attention to the designations for the pins (E,B,C for emitter, base, collector) on the device. Pin 1 is the emitter, pin 2—in the middle—is the base, and pin 3 is the collector. The anode of a digit must be able to supply enough current for all eight segments of the digit simultaneously. The datasheet for the TDCR1050M shows that each of the eight segments of each digit could use as much as 25 mA, although you won't have that much current flow for this experiment. A microcontroller pin would not be able to supply enough current to illuminate all of the segments of a digit, but a PN2907 transistor will provide at least 200mA if needed. Because the transistor is driving the positive voltage to the display, it is referred to as the "high-side" driver for the display. You may think of the sink driver as the "low-side" driver for the display.

Rather than try to deal with low voltage to enable current flow through each PNP transistor, we will add one more device: a 3-to-8 decoder with active-low outputs. Three input pins select one of eight outputs to be driven low. Each one of these outputs will be connected, through a series resistor, to the base of one high-side driver transistor. Altogether, eleven pins of the microcontroller will be used to drive the display: three to select the digit to turn on through the decoder and high-side driver transistors, and eight more to turn on the segments of the

selected digit. A full schematic of the wiring for the display and keypad is shown in Figure 6.

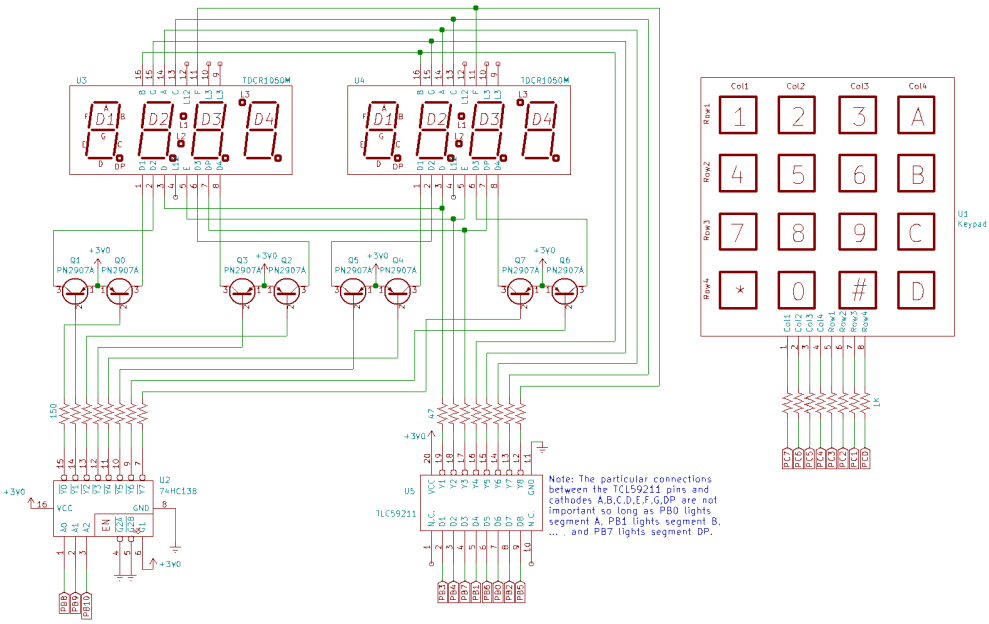


Figure 6: Schematic for lab experiment wiring

It can be difficult to follow all of the connections as they cross over each other. A second schematic, with most of the wires replaced by symbolic connections is shown in Figure 7. That should offer slightly more clarity in how the connections are made. A schematic is normally intended for a high-level understanding of the logical connections, and has few details about the physical layout of components. Nevertheless, some nuances of the physical construction are preserved in these schematics. In particular, note that the transistors are numbered, from left to right, Q1, Q0, Q3, Q2, and so on. This is intentional since that is the way they will be placed on the breadboard.

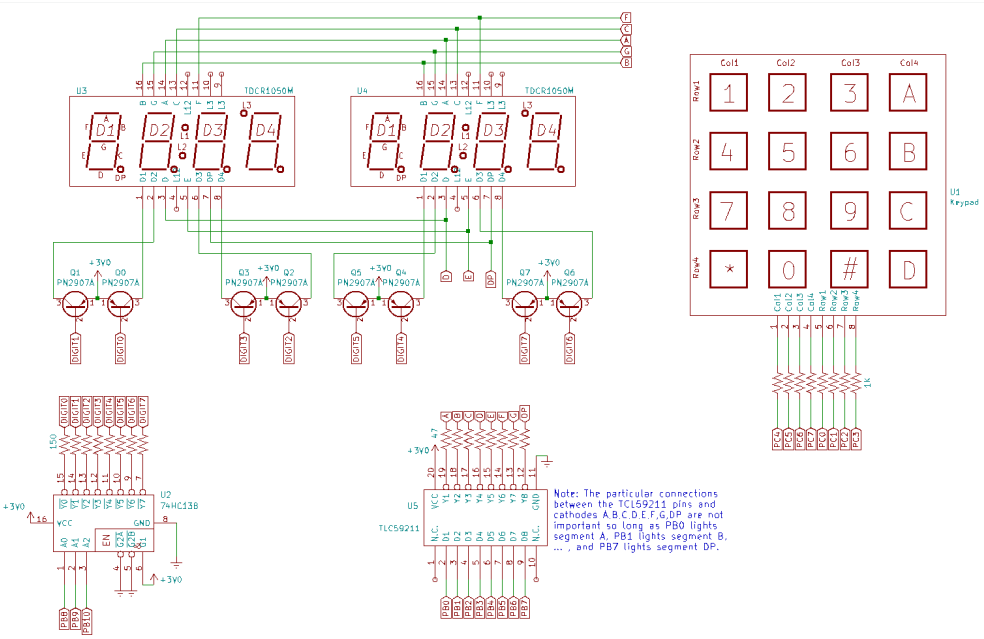
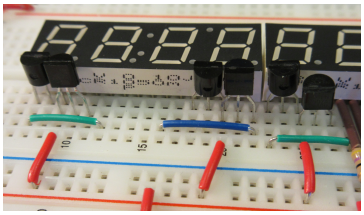


Figure 7: Simplified schematic for lab experiment wiring

1.5 Tips for wiring the circuit

You'll be assembling two 4-digit display modules, driven by 8 PN2970A transistors, a TLC59211 sink driver, 8 resistors between the sink driver and LED cathodes, a 3-to-8 decoder, and 8 more resistors between the decoder and the transistors. Another eight resistors will be used to connect the keypad to the microcontroller. With some planning, it will be possible to assemble this quickly and reliably. Here are several suggestions to make it as easy as possible...

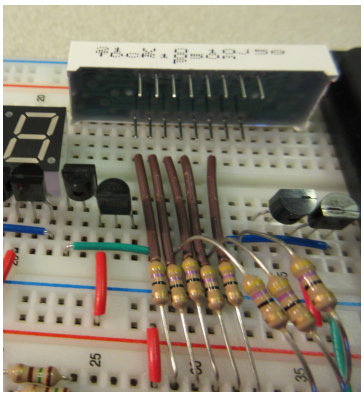
1.5.1 Place the transistors next to the displays



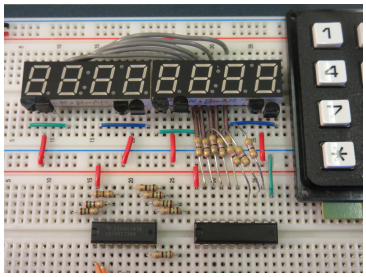
The collector of a PN2907A transistor must be connected to each of pins 1, 2, 6, and 8 of both display modules. Notice that these pins are all on one side of the display, and that two of them are near the ends of the row of pins. There are no connections to the left or right of the connector, but there is a lot of space taken up by the displays. Rather than run long wires from the transistors to the displays, put the collector of one transistor on pin 1 and the collector of another transistor on pin 8. The other pins of each transistor are placed to the left and right, respectively. Add another two transistors further out by combining the emitters on the same breadboard row. Finally,

connect the collectors of the outermost transistors to pins 2 and 6. Looking from left to right, the first backward-facing transitor will connect power to D2 (pin 2). The next forward-facing transistor will connect power to D1 (pin 1). The next transistor will connect power to D4 (pin 8), and the fourth transistor will connect power to D3 (pin 6). Note the alternating backward-facing, forward-facing manner in which the transistors are placed. The base (middle pin) of each of the transistors is what will be driven, through a 150Ω resistor, by the decoder.

1.5.2 Wire the resistors directly to the display

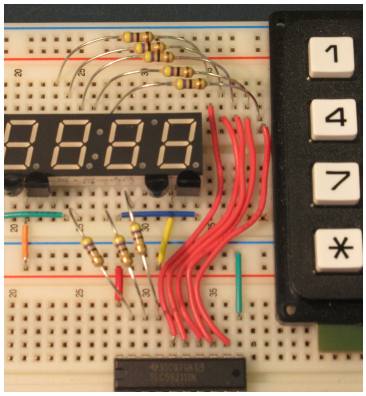


Five of the LED cathodes are on the "top" side row of connectors (pins 16,15,14,13,11) of the display, and three of them are on the "bottom" row of connectors (pins 3,5,7). Each of the eight cathodes must be connected to an output of the TLC59211 sink driver via a series 47 Ω resistor. If you have bulk 22 gauge copper wire, you can remove a suitable amount of insulation and put it on each of the five resistors to keep them from touching the pins they run near. Run the resistor leads between the pins below the display. Place the TLC59211 on an adjacent breadboard, in the right position to directly connect the resistors. The other three resistors are easy to wire between pins 3,5,7 and the sink driver. Place the resistors between the cathodes and the TLC59211 outputs in the most convenient way possible. Connect the wires between the STM32 Port C and the TLC59211 so that PB0 lights segment A, PB1 lights segment B, ... , PB6 lights segment G, and PB7 lights segment DP.



If you are not able to insulate resistor leads in this way, leave some space on the right side of the rightmost display module, connect the resistors on the top side of the display module, and run wires from the other side of the resistors to the TLC59211. It's your choice. Either way will work well. Neatness of connections will avoid a great deal of problems when you reach the point of debugging. A maze of wires that

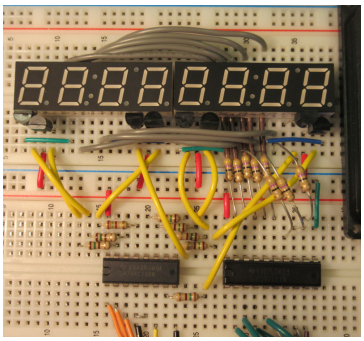
loop far over the breadboard will not only block the LED segments, it will be difficult to diagnose, and be prone to damage when the wires are snagged on objects while in transport.



1.5.3 Chain the cathodes from one display to the other

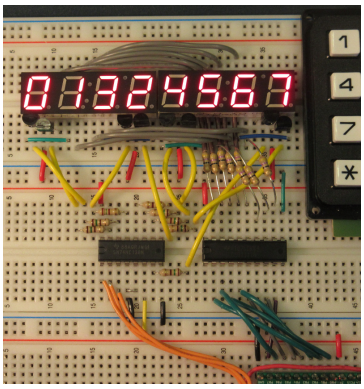
The eight connections to the cathodes of display on the right should be made to the same eight pins on the display on the left. Use eight wires to connect them together. I.e., connect pin 16 on the left display to pin 16 of the right display, pin 15 to 15, etc. The pins to connect together on the top side of the displays are 16, 15, 14, 13, and 11. The pins on the bottom side of the displays are 3, 5, and 7.

1.5.4 Connect resistors to the decoder



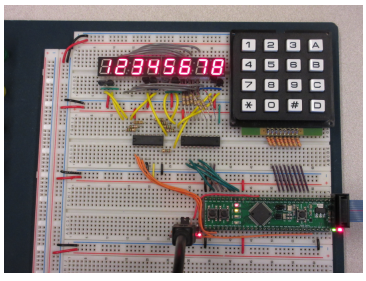
The 150Ω resistors should be placed directly on the 74HC138 decoder. If you want to be tidy, you might cut the leads to minimize the amount that they stick out of the breadboard. If you do so, make sure that leads of every resistor is wide enough to span four breadboard holes. A width of only three would mean, for instance, that the resistor on the Y2 output would connect to the V_{CC} pin of the chip. Wires should be run from the other ends of the resistors to the bases of the transistors. Connect output Y0 through a 150Ω resistor to the base of the PNP transistor connected to pin 1 of the left display. Note that output Y7 is on the bottom side of the 74HC138 decoder on pin 7. Pin 8 is the ground connection. The decoder's select lines, A0, A1, and A2 (pins 1,2, and 3, respectively) should be connected to PC6, PC7, and PC8. Pins 4, 5, and 6 should be connected to GND, GND, and V_{CC}, respectively. These three pins are the *enable* pins for the decoder. Unless all three of these pins are enabled, **none of the decoder outputs will be asserted** (driven low).

1.5.5 Check your wiring



Look ahead to Section 2 of this lab document to find out how to set up the main.s template and obtain the autotest.o module. The autotest.o module for this experiment also has a wiring checker. Uncomment the **`bl check_wiring`** call in main.s. When first started, the digits **`01234567`** should appear on the seven-segment displays. The decimal point on the leftmost digit should illuminate for 0.25 seconds, then turn off and the decimal point on the next digit should illuminate for 0.25 seconds. The decimal point should cycle through all of the digits in two seconds and continue again with the leftmost. The picture to the left shows a situation where the bases of two transistors are connected to the wrong decoder output resistors. Instead of **`01234567`**, the display shows **`01324567`**. An easy mistake to make as well as to correct.

As the decimal point illuminates, make sure that the other segments of each digit remain at the same brightness. If the other segments of a digit grow dimmer, it may be that the driver transistor for that digit is *backward*. A BJT transistor normally amplifies a small base-emitter current with a large collector-emitter current, but a BJT can also work backward so that it amplifies a small base-collector current to a larger emitter-collector current. The amplification is not as high, and it limits the current provided to a digit. The more segments per digit are lit, the dimmer they are. Carefully note the orientation of the transistors connected to the D1, D2, D3, and pins of each display.



Once the display is showing the correct digits and the decimal point is repeatedly moving from left to right, try pressing buttons on the keypad. Each button will cause a corresponding letter to be shifted onto the display to the rightmost digit, and the other seven digits will shift left. The asterisk is represented with a Greek Xi (Ξ), and the **octothorpe** is represented with an "H". (You think I'm inventing a new word here? Go look that up.) If you press the '1' button,

and a new 'A' gets shifted in on the right, it means you've switched the Row1 and Row4 wires on the keypad. If you press the '3' button, and a new '0' is shifted in on the right, it means that the wires to Row2 and Row3, as well as the wires to Col1 and Col4, are switched. By pressing buttons and looking at the outcome, you can determine which wires are incorrect.

2.0 Experiment

Create a project in SystemWorkbench called "lab5", and delete the automatically-generated main.c file. Create a main.s file in the src directory. In order to expedite setting up the framework for the code you will write, as well as getting all of the constants you will need throughout the lab, a *template* file for main.s is provided for you. You should download main.s and copy the contents into your own lab5/src/main.s file to get started.

This experiment uses a pre-compiled object file that contains test functions to automatically test the subroutines you write for this lab. It is called autotest.o. To incorporate it into your System Workbench project, follow the instructions in the ECE 362 References page for Adding a pre-compiled module to a project in System Workbench. It provides an interactive serial interface in the same manner that you have seen in previous lab experiments. You can use it to test the operations you write.

For this lab experiment, the autotest.o module also contains a **`check_wiring`** subroutine to test that the LED displays and keypad are wired properly. It also contains duplicate subroutines for everything you will implement in the lab (except the ISRs). For any normal subroutine, you can preface it with **`good_`** to try a pre-written subroutine that works correctly. If, for instance, you are having difficulty with the **`enable_ports`**, you can replace the call with **`good_enable_ports`**. You won't get get a score for any work you do while you are using any of the good_ subroutines. They're only a temporary aid for development.

2.1: enable_ports: Enable GPIO ports

Write a subroutine called **`enable_ports`** that does the following:

- Enables the RCC clock to GPIOB and GPIOC without affecting any other RCC clock settings for other peripherals
- Configures pins PB0 – PB10 to be outputs
- Configures pins PC4 – PC7 to be outputs
- Configures pin PC9 to be an output
- Configures pins PC0 – PC3 to be inputs
- Configures pins PC0 – PC3 to be internally *pulled low*

As usual, you should not alter the configuration for any other pins. For instance, if you were to improperly alter the MODER setting for PC12, the serial port would stop working.

2.2: Basic Interrupt Service Routine (ISR)

Write an interrupt service routine (ISR) for Timer 6 (TIM6). Look up the exact name of the ISR in startup/startup_stm32.s and copy it into the recommended spot in your main.s. (Yes, it has a long, strange name that you will understand the reasons for later.) Designate it as a Thumb function, and make it global. It should do the following things:

- Normal ISRs must *acknowledge* an interrupt so that it will not be immediately reinvoked. To acknowledge a basic timer interrupt, you must write a zero to the UIF bit of the status register (SR) of the timer. Sometimes, interrupts must be acknowledged by writing a '1' bit as you saw for the EXTI subsystem. In this case, you write a '0' bit to the UIF bit to clear it, and write all other bits to '1' to avoid clearing anything else. In other words, you write the bitwise inverse of TIM_SR_UIF, which is ~TIM_SR_UIF. There will be no other bits set in the TIM6_SR register, so you may simply write 32 bits of zeros to the TIM6_SR register. It will do what you need and not cause any other problems. Be prepared to be more selective with peripheral status registers in future lab experiments.
- Toggle the PC9 bit. To do so, you should check the current value of bit 9 of the ODR. If it is set, use the BRR register to turn it off. Otherwise, use the BSRR register to turn it on. Avoid writing

directly to the ODR. Multiple ISRs will modify pins of GPIOC, so it is important to use *atomic* operations.

This is the simplest way that you can demonstrate any timer: Write the simplest possible ISR, and blink one LED.

2.3: setup_tim6: Configure a basic timer

Now that you have written a simple ISR for the Timer 6 interrupt, you can configure timer 6 to invoke that interrupt. Write a subroutine named **`setup_tim6`** that does the following: (See the lecture notes for an example.)

- Enables the RCC clock to Timer 6. It is up to you to determine how to do so. You might look at Section 7.4 of the Family Reference Manual to find which bits of which register (usually, one of AHBENR, APB1ENR, or APB2ENR) to set. You can also look it up in the I/O Register browser in SystemWorkbench. Practice doing so. You will need to know how to do this for a lab practical.
- Configure TIM6_PSC to *prescale* the system clock by 48000. (You should know, by now, that you should not write the value 48000 to the PSC register to do this.) This will divide the 48 MHz system clock by 48000 to send a 1 kHz clock to the free-running counter of the timer.
- Configure the Timer 6 auto-reload register (TIM6_ARR) to have a counting period of 500. With an input clock of 1 kHz, this will cause a timer update event to occur every 0.5 seconds. Since we intend to raise an interrupt that will run the ISR you just wrote, it will cause the PC6 LED to toggle every 0.5 seconds (or blink at a rate of 1 Hz).
- Configure the Timer 6 DMA/Interrupt Enable Register (TIM6_DIER) to enable the UIE flag (use the symbol TIM_DIER_UIE for this). This will enable an update interrupt to occur each time the free-running counter of the timer reaches the ARR value and starts back at zero.
- Enable Timer 6 to start counting by setting the CEN bit in the Timer 6 Control Register 1. (Set TIM_CR1_CEN in TIM6_CR1.)

- Enable the interrupt for Timer 6 in the NVIC ISER in a manner similar to how you did in lab 4.

Once you complete this subroutine and run it, you will see the PC9 (Blue) LED blink at a rate of 1 Hz.

2.4 A display multiplexing and key scanning application

Once you've successfully written a timer ISR, more interesting applications are easy to implement. Rather than add them to the ISR you just wrote, you'll write an ISR for a second timer and a configuration subroutine that will turn off Timer 6. This will allow the code you just wrote to be tested independently.

The application to be built for this lab is a simple number-entry-and-display system. You will use the concepts for this system, as well as the hardware it uses, for future lab experiments.

2.4.1 Data structures for the application

Three very simple global "data structures" will be used to keep track of the values needed for this application. They are already defined at the bottom of the main.s template file.

- **`col`** is a single-byte integer, with a value ranging from 0 – 7, that indicates the digit presently being displayed. **AND** the lower two bits indicate the row that is being scanned. For purposes of these subroutines, digits will be numbered, left-to-right 0 – 7, and keypad columns will be numbered, right-to-left 0 – 3. For instance, if the value of **`col`** is 6, that would mean that digit 6 (second from the right) is illuminated, and that column 2 (containing buttons 2, 5, 8, and 0) is being scanned.
- **`disp`** is an 8-entry array of single-byte values, each of which represent the characters to display for each digit. The entries for this array should be pre-initialized to the following values:

```
disp: .string "Hello..."
```

This string will be displayed on the segments on start.

- **`hist`** is a 16-entry array of single-byte values, each of which represents the history of scanned buttons of the keypad. Elements 0 – 3 represent the histories of the top row of buttons (1, 2, 3, A). These 16 bytes can be initialized with the **`.byte`** directive.

These variables are created and initialized at the bottom of your main.s file.

2.4.2: show_char: Display a single character

Implement, in ARM Cortex-M0 assembly language, the following subroutine that will select a digit to light using PB8, PB9, and PB10 and also output the 8-bit pattern to display on that digit on PB0 – PB7. A table provided for you gives the mapping from the character number to the segments of the LEDs to illuminate. Such a data structure could arguably be called a *font* since it shows the elements of the data to display. You can call it exactly that. The **`font`** array is in read-only memory (i.e., in the text segment) at the bottom of the text file. It includes most of the printable ASCII character set, although some characters are not possible on a 7-segment LED (e.g., "m" and "w").

To implement the subroutine, the pattern should be read from the **n**th element of the **`font`** array. Since all of the Port B output pins are being used for the display, you may simply output the combined value to the ODR. Note that, as a defensive coding practice, you should ensure that the offset value is in the range of 0 – 7.

```
void show_char(int n, char c)
{
    n = n & 7;
    GPIOB->ODR = (n << 8) | font[c];
}
```

Note that **`font`** is an array of single-byte values. You should use the single-byte load instruction when reading its values.

2.4.2.1: fill_alpha: Show sequential letters

Once you implement **show_char**, you my uncomment the call to **fill_alpha**, which is provided for you. It will call **show_char** with the values 'A' through 'H' to show the series on the display. You may use this for debugging. Once it is working, show your TA to confirm that you have completed this much of the lab experiment. You will not receive credit for the remainder of the lab if you cannot demonstrate this much in lab.

2.4.3: drive_column: Update the keypad column being scanned

Implement the following subroutine that updates the column of the keypad that has a logic high applied to it. Remember that the column being scanned is the least two significant bits of the **col** variable. Not all of the pins of Port C configured for output are used for the same thing. One of them is for blinking PC9. You should not simply write to the ODR to implement this subroutine. Use the BSRR register to update only the bits 4 – 7, like this:

```
void drive_column(int c)
{
    int row = offset & 3;
    GPIOC->BSRR = 0xf00000 | (1 << (c + 4));
}
```

2.4.4: read_rows: Read the row values

Implement the following function to examine the IDR of Port C and return the 4-bit reading of the columns of the keypad.

```
int read_rows()
{
    return GPIOC->IDR & 0xf;
}
```

In other words, bits 3 – 0 of R0 should hold the four bits read from PC3 – PC0.

2.4.5: update_history: Update the history bytes for one column

Implement the following function that updates the **`hist`** array bytes corresponding to the column being checked. Note that the **`c`** variable is expected to be passed in as the first parameter to indicate the column, and **`rows`** is the second parameter that represents the four row bits.

```
void update_history(int c, int rows)
{
    c = c & 3;
    for(int i=0; i < 4; i++)
        hist[4*c+i] = (hist[4*c+i]<<1) + ((rows>>i)&1)
}
```

As with anything you do in this course, there are actually many ways of simplifying and implementing a subroutine, so a few variations are offered for the sake of illustration. Pick the one you like most.

```
void update_history(int c, int rows)
{
    c = c & 3;
    for(int i=0; i < 4; i++) {
        hist[4*c+i] = hist[4*c+i] << 1;
        if ((rows & (1 << i)) != 0)
            hist[4*c+i] |= 1;
    }
}
```

```
void update_history(int c, int rows)
{
    c = c & 3;
    char *hcol = &hist[4*c];
```

```
    for(int i=0; i < 4; i++) {
        hcol[i] = hcol[i] << 1;
        if (((rows >> i) & 1) != 0)
            hcol[i] += 1;
    }
}


void update_history(int c, int rows)
{
    c = c & 3;
    char *hcol = &hist[4*c];
    char tmp = *hcol <:< 1;
    char carry = (rows>>0) & 1;
    *hcol = tmp + carry;
    hcol ++;
    tmp = *hcol << 1;
    carry = (rows>>1) & 1;
    *hcol = tmp + carry;
    hcol ++;
    tmp = *hcol << 1;
    carry = (rows>>2) & 1;
    *hcol = tmp + carry;
    hcol ++;
    tmp = *hcol << 1;
    carry = (rows>>3) & 1;
    *hcol = tmp + carry;
}
```

That last form can be written using only four registers, and no branches. Naming one of the variables "carry" is a hint. If you've reached the point that you can write the subroutine this way, then you are a truly competent assembly language coder. (There is no shame if you're not.)

2.4.6: Timer 7 ISR

Write an ISR for timer 7. You should look up the exact name for it in startup/startup_stm32.s, declare it to be a Thumb function, and make it global. The ISR should first acknowledge the timer interrupt (similar to what you did for the timer 6 ISR). Then, it should do the following things to invoke the subroutines you just wrote:

```
int rows = read_rows();
update_history(col, rows);
char ch = disp[col];
show_char(col, ch);
col = (col + 1) & 7;
drive_column(col);
```

You may, of course, omit the temporary variable used between the calls to read_rows() and update_history(). In assembly language, you can just use the R0 register.

Remember that **`col`** is a single-byte integer. You should not use **`ldr`** or **`str`** to access it. Use the single-byte load and store instructions.

Here, read_rows() is called, and update_history() is called. The col value is incremented, and then drive_column() is called. This is a non-intuitive order of events, and it often confuses students. The reason we do not call drive_column() and then immediately call read_rows() is because it sometimes takes a little time for current to flow into the row of buttons and *settle out* to be read properly on the input pins. It doesn't always happen, but when students energize the column and immediately try to read the rows, and it *doesn't* work, then it's a Bad Day ™.

You don't want that to happen to you, and neither do we. Here, you don't get to choose. You must call the functions in the order shown above. Otherwise autotest won't count it as correct.

2.4.7: setup_tim7: Configure timer 7

Write a subroutine to enable and configure timer 7. This will be similar to the subroutine you wrote to configure timer 6. Remember that timer

6 was written as a simple test, and it interferes with everything you've written since then (because it updates PC6). This is the point where you must disable timer 6. Here are the things you need to do:

- Enable the RCC clock for TIM7. You must look up how to do this so that you will be prepared for a lab practical.
- Set the Prescaler and Auto-Reload Register to result in a timer update event exactly once per millisecond. (i.e., a frequency of 1 kHz)
- Enable the UIE bit in the DIER.
- Enable the Timer 7 interrupt in the NVIC ISER.
- Set the CEN bit in TIM7_CR1

Note that another common mistake made by students is to set the TIM7_ARR register to zero. When you do that, the timer won't do anything. That's worth a bigger warning...

If you set the Auto-Reload Register of a timer to zero, the timer won't do anything. If you're tempted to do that, divide the prescaler setting by a certain amount and multiply the desired timer counter period by the same amount.

Debugging tip: It is often helpful to slow down the rate of interrupts to *human speed* to better understand what is going on. For instance, if you set the timer 7 rate to one interrupt per second, you'll be able to see exactly what is happening. You should see individual digits displayed for one second at a time on the LED displays. You'll be able to press buttons at exactly the right time to be detected by `read_rows`.

2.4.8: wait_for: Wait for a single keypad button event

The subroutines written above will be used to show digits on the display and scan the keypad. They are called by the timer 7 ISR. So far, there is nothing to *interpret* or use the keypad events. Now you will

write a function called **`wait_for`** that will *wait for* a key to have a particular history state and return the key number in R0.

The keypad columns are scanned from right-to-left, where the rightmost column is column 0 and the leftmost column is column 3. Rows are interpreted from bottom-to-top. Each column occupies four contigous bytes in the history array. The (hexadecimal) offset of each button in the history table is as follows:

```
Buttons:                Offsets:
+---+---+---+---+       +---+---+---+---+
| 1 | 2 | 3 | A |       | f | b | 7 | 3 |
+---+---+---+---+       +---+---+---+---+
| 4 | 5 | 6 | B |       | e | a | 6 | 2 |
+---+---+---+---+       +---+---+---+---+
| 7 | 8 | 9 | C |       | d | 9 | 5 | 1 |
+---+---+---+---+       +---+---+---+---+
| * | 0 | # | D |       | c | 8 | 4 | 0 |
+---+---+---+---+       +---+---+---+---+
```

For instance, if the '8' button is pressed, the function should return the value 0x9 (an offset of one from the start of column 2: 2*4 + 1 = 9). When the '*' button is pressed, the function should return the value 12 (hexadcimal 0xc) since it is in column 3, row 0 (3*4 + 0 = 12).

One way to do this would be to have a function that continually looked at the history values searching for a 0x01 like this:

```
int wait_for(int state) // the naïve way
{                       // for example only
    for(;;) {
        for(int i=0; i < 16; i++)
            if (hist[i] == state)
                return i;
    }
}
```

When **wait_for** is called with an argument of 1, this would certainly detect the case where a button is pressed, but it would be very inefficient. The main loop of the program calls it repeatedly and runs continuously until it detects a button press. Because the main program calls it repeatedly, it would also report a single key press multiple times. You'll have an opportunity to try it below. You'll see how badly it works.

Instead of the naïve approach, we recognize two things:

- The history array entries will only be updated by an interrupt. It is only necessary to check the history array after each interrupt.
- Every key that is pressed must be released. If we wait for each key to be released before we accept another key pressed we can avoid the situation where multiple interrupts occur while a key is pressed for a first time.

Therefore, the subroutine should, instead, wait for the conclusion of an interrupt, then scan the history bytes for the requested state. How can we Wait For an Interrupt? By using the WFI instruction:

```
int wait_for(int state)
{
    for(;;) {
        asm volatile ("wfi" : :);   // wait for an interrup
        for(int i=0; i < 16; i++)
            if (hist[i] == state)
                return i;
    }
}
```

The **asm volatile ("wfi" : :);** is simply the means by which we can embed a WFI instruction into a C program. (See lecture 09 on embedded C for more information.) When writing an assembly language program, you can just type "wfi". When the CPU executes the WFI instruction, it will put the microcontroller into *sleep mode*. This is a low-power state that is useful for reducing power consumption. Although the CPU is asleep, the peripheral subsystems that have been enabled continue to run. The CPU is awakened from sleep mode by an interrupt to run the ISR. When the ISR returns, the code following the

WFI instruction is executed. In this way, the code to evaluate the history array can only happen after an ISR completes.

The important thing to recognize is that this subroutine runs in the context of the *main program* of the application, and it does not return until an interrupt has happened, and the conditions of the history array constitute a button press.

2.4.9: shift_display: Shift all display entries left

Implement a subroutine that will take the seven "rightmost" elements of the **`disp`** array and shift them left by one position. Basically, do the following:

```
void shift_display(void) {
  for(int i=0; i < 7; i++)
    disp[i] = disp[+1];
}
```

Remember that the elements in the **`disp`** array are one byte each. You should use the single-byte load and store instructions to move the data. It only needs to update the **`disp`** array. It does not need to change any GPIO pins.

Once **`shift_display`** works, another subroutine provided for you — **`handle_keypress`** — will wait for a key press, shift the display, look up the character for the key, insert it at the right side of the display, and wait for the key to be released.

What character corresponds to any particular button? We'll use another read-only data structure to specify this information. The **`keymap`** array is a 16-entry table of the characters produced for a particular button. For instance, the '9' button corresponds to entry 5 in the history array. When it is pressed, **`wait_for`** will return 5. To convert it into a character, look up element 5 of the **`keymap`** array.

See the implementation of **`handle_keypress`** to learn more.

3 Observations and Thoughts

Once your program works, and you can reliably press buttons and see their values shifted onto the display, you should first congratulate yourself on a large job well done.

There is, however, a bit more thinking and learning to do.

3.1 Unrelated interrupts

When the **`wait_for`** function waits for an interrupt, it assumes that the only thing producing an interrupt is timer 7. Recall that there are many forms of interrupts possible from the many timers and other subsystems in the STM32. If, for instance, you were to configure timer 1 to produce an interrupt every 10 microseconds, then **`wait_for`** would be *awoken* 101 times per millisecond to check the history array. This would be rather inefficient.

You can try it. Enable Timer 1, and create a simple ISR that only acknowledges the interrupt. (This will be good practice for the lab practical exam.) You might also change timer 6 so that it produces an update event at a much higher frequency.

In the future, you will use the keypad and display for other applications, and that will happen with other interrupts enabled and occurring. What should be done about this?

First, it is necessary to recognize that the exercises in this lab were constructed to illustrate different elements of timer configuration, interrupt handling, and waiting for interrupts in the main program. Like many things you implement over the span of your career, this program only works for the limited set of circumstances it was designed for. Outside of those circumstances, it is incorrect.

The second thing to realize is that implementing a better solution would be too difficult to accomplish easily in assembly language. (This lab was difficult enough, right?) In future lab experiments, you will be using the C language, in which you can more easily express complex data operations.

A better way to handle detection of button presses would be to do the detection in **`update_hist`** and have it write an entry into a circular queue of button events. Meanwhile, **`wait_for`** would wait on the head of the circular queue to change from an *invalid* entry to a *valid* entry to indicate a button press. Once an entry was read from the queue, **`wait_for`** would then change that entry back to invalid and wait for the next entry in the queue to become valid. There are still circumstances that this will not work (for instance, what happens if buttons are pressed and detected at a faster rate than they are being read from the queue). As the circumstances change, as long as you are aware of them, you will be able to adapt the system to avoid problems.

4 Submit your postlab results

Once your lab experiment works to your satisfaction, autotest should tell you how many parts work correctly on the serial terminal. The autotest module will display a completion code similar to that of some previous labs. This code is a cryptographic confirmation of your completion of the lab experiment as well as your identity and your score. (Remember to change the "login" variable from "xyz" to your login.) It is normal for the code to change every time you run it. Choose one of the codes and carefully enter all 32 characters into the post-lab submission for this lab experiment.

For this lab, and most to follow, you must also submit the program that you wrote so that it can be checked by the course staff. Either upload the file or copy it from SystemWorkbench and paste it into the text box. Make sure that your entire program is shown there.

Lab Evaluation Checklist

Normally, we'll have a checklist of things that you should review before going into your evaluation. There will be no evaluation for this lab experiment.

- [ ] Did you demonstrate your wiring and early code to your lab TA?
- [ ] Did you submit your early code?
- [ ]

- [ ] Did you clean up your lab station and log out?
- [ ] You did not share your microcontroller with anyone else, right? We will know if you did.