Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.115 Microprocessor Project Laboratory Spring 2022

Laboratory 1
Getting information into and out of the microcontroller

Issued: February 9, 2022 Due: February 23, 2022

GOALS: Get familiar with your R31JP and PSoC development systems
Learn to communicate with the microcontroller from a personal computer
Use a custom keypad to provide data to the microcontroller
Make simple calculations and data manipulations using the microcontroller
Display results on the PC screen and on the development system's LED bank

PRIOR to starting this lab:
READ: INTEL MCS-51 Manual: p.1-3 – 1-22; 2-3 – 2-20; 3-3 – 3-9; 3-13 – 3-20;
and SKIM (just be aware of) pages 2-21 – 2-75. READ: 74C922 Data Sheet.
READ: Lab 1 INTRODUCTION document, sent separately.
SKIM the R31JP user's manual, and READ in Yeralan: Chapter 1, and sections 2-1 – 2-4.
Complete the kit familiarization in lecture – do NOT use your kit without attending the lecture.

EXERCISE 1: See the shiny lights.

First, READ the Laboratory 1 INTRODUCTION document handed out in class last week. Make sure that you have programmed MINMON into the R31JP monitor ROM as described in our kit familiarization lecture. Connect your R31JP board to the RS232 port of a Windows 10 compatible personal computer using the DB-9 serial cable available on the benches in the 38-600 laboratory. Or, if you are working at home, use your Kable to connect your laptop or desktop to the R31JP.

Generally in 6.115, you will write programs from "scratch". For the first two exercises in this laboratory, let's break with this plan and experiment with some "canned" code. Let's try a program that will turn on one of the lights on the LED bank on your R31JP. This LED bank is wired to Port 1 on the 8051 microcontoller. The LED bank contains 10 individual lights. Eight of these lights are wired to the 8 lines of Port 1. Two of the lights are not connected to anything. Please assemble and run the following assembly language program using AS31:

Test Program:

```
; This little program turns an LED on.

mov P1, #00h                    ; Clear the LED bank
mov P1, #01h                    ; Turn on a single light
loop:
   sjmp loop
```

This very exciting program should light a single light on the LED bank. You now should know which light is connected to the least-significant bit (LSB) on Port 1.

Please do the following:

- Assemble the test program above using AS31. Download the resulting HEX file to the R31JP using your Kable and TeraTerm. Run the program using either the MINMON "G" command or the MON/RUN and reset switches on the R31JP. What do you see on the R31JP Lights?
- Explain the purpose of the loop in the test program above.
- Write and test a program to verify which light is connected to Port 1's most-significant bit (MSB).
- Write another short program (5 lines or less) that is functionally identical to the test program, but which uses the `clr` and `setb` commands to activate the light. You may use other commands as well, but be sure to use the `clr` and setb commands in your program.
- Experiment with the light bank to see if you can make an interesting static (unchanging) pattern, e.g., every other light off.
- Next, consider the problem of making a visibly flashing light or an interesting dynamic pattern on the lights (such as left-to-right and back again), e.g., from the classic TV show or movie of your choice, such as a "Knight-Rider" pattern, "Cylon-eye" from Battlestar Galactica, or "Launch-bay lights" from Buck Rogers. Any interesting pattern is fine. Think about whether you could accomplish this with the `nop, ljmp,` and `clr` commands. What problems might be encountered in the attempt to make a flashing light strictly with `mov` and `nop` commands? Can you make things work using the `djnz` command?

EXERCISE 2: Serial communications with a personal computer.

The 8051/8052 microcontrollers contain serial communications hardware that can be used to communicate with any terminal or personal computer that supports RS232-style communications. In this first exercise, let's develop a simple "typewriter" program.

Here's the main body of our program, which you should type into a file:

Program Segement 1:

```
; The main loop or body of our typewriter program:

.org 00h               ; power up and reset vector
   ljmp start          ; when the micro wakes up, jump to the beginning of
                       ; the main body or loop in the program, called "start"
.org 100h              ; and located at address location 100h in external mem
start:
 lcall init            ; Start the serial port by calling subroutine "init".
 loop:                 ; Now, endlessly repeat a loop that
   lcall getchr        ; <- gets a character from the PC keyboard
   lcall sndchr        ; -> and then echoes the character to the PC screen
 sjmp loop
```

To complete the typewriter program, you need to provide three subroutines, `init, getchr, and sndchr`. These three routines, or some functionally similar code, are critical tools anytime you wish to

communicate back and forth from your microcontroller using a serial connection. These three routines are used to communicate with a personal computer, for example, in the MINMON monitor code that runs on your R31JP board when you reset or apply power to the board. Let's look at incomplete code segments for each of these three subroutines. You fill in the necessary constants to make things work. Your readings have revealed to you the initial or "power-up" state of the SMOD bit in the PCON special function register, which you will need to know to complete Program Segment 2.

Program Segment 2: (Init routine)

```
init:
; set up serial port with a 11.0592 MHz crystal,
; use timer 1 for 9600 baud serial communications
   mov   tmod, #       ; set timer 1 for auto reload - mode 2
   mov   tcon, #       ; run timer 1
   mov   th1,  #       ; set 9600 baud with xtal=11.059mhz
   mov   scon, #       ; set serial control reg for 8 bit data
                       ; and mode 1
   ret
```

Program Segment 3: (Getchr routine)

```
getchr:
; This routine "gets" or receives a character from the PC, transmitted over
; the serial port.  RI is the same as SCON.0 – the assembler recognizes
; either shorthand. The 7-bit ASCII code is returned in the accumulator.

   jnb  ri, getchr        ; wait till character received
   mov  a,  sbuf          ; get character and put it in the accumulator
   anl  a,  #7fh          ; mask off 8th bit
   clr  ri                ; clear serial "receive status" flag
   ret
```

Program Segment 4: (Sndchr routine)

```
sndchr:
; This routine "sends" or transmits a character to the PC, using the serial
; port.  The character to be sent is stored in the accumulator.  SCON.1 and
; TI are the same as far as the assembler is concerned.

   clr  scon.1            ; clear the ti complete flag.
   mov  sbuf,a            ; move a character from acc to the sbuf
txloop:
   jnb  scon.1, txloop    ; wait till chr is sent
   ret
```

Program segments 3 and 4 are complete. Program segment 2, the initialization code, will require you to fill in four constants to appropriately activate the timer and serial buffer.

Please consider and do the following:

- Complete program segment 2 with register constants for 9600-baud communication.
- Assemble the complete program consisting of code segments 1-4, and test your code by connecting the R31JP to a PC. Use a communications program like TeraTerm in Windows on the PC to send and receive characters to and from the microcontroller.
- How could program segment 1 be rewritten to use less program memory? There are at least three changes that could be made. Do these changes make the code easier to understand, harder to understand, or leave it about the same?
- In your lab report, also include the register constants for program segment 2 that you would have used if we had instead wanted 2400-baud communication. You do not need to recompile your actual program for 2400 baud operation. Simply note the constants you would have used for 2400 baud communication in your lab report.

EXERCISE 3: Make an ASCII table and improve the typewriter

Characters like the letter "a" or the number "2" that you see displayed on the computer screen when you interact with the microcontroller are actually represented as binary numbers according to an accepted standard or code. There are a number of popular codes. Our microcontroller firmware and the PC terminal emulator exchange data using the *American Standard Code for Information Interchange* or *ASCII*. In the ASCII system, English letters and Arabic numerals are represented by a 7 bit code. This code permits the representation of 128 characters, which include the entire alphabet in both upper and lower case, single digit numerals, and some control commands that cause terminal programs like TeraTerm to do something, e.g., carriage return to the beginning of a line or drop down to a new line (line feed).

With some improvement, the little PC "typewriter" program in Exercise 2 can be used to help understand the ASCII code system. Make modifications to program segment 1 to accomplish the following:

- Add a single line to Program Segment 1 that causes the ASCII code for each character typed to not only be echoed back to the PC (by the routine `sndchr`) but also to be displayed on the LED bank connected to Port 1.
- Implement an "auto-wrap." After the typist has entered 65 key strokes, have the microcontroller automatically issue a carriage return (back to the beginning of the line) AND a line feed (drop down to the next line). Add a subroutine called `crlf` to your program to provide this operation. The ASCII code for a linefeed operation is the decimal number 10. The ASCII code for a carriage return operation is the decimal number 13. Use the `djnz` command to count 65 key strokes.
- Use your modified typewriter program to fill in the missing characters on the ASCII table attached to the end of this lab, a sort of periodic table for computer nerds. The first two rows of control codes/special characters have been filled in for you, along with a few other characters to give you the point. You will notice patterns in the table. Please do not guess codes and do not look them up in a reference until you have filled in the table using your own program. Check each one with a key press. Notice that the table provides each code as a decimal, binary (low and high nibble), and hexadecimal number. Please use this as an opportunity to make sure that you understand not only the ASCII code, which you will use all term, but also the different number systems. Keep a copy of your ASCII table for future reference, and include a completed copy in your lab report submission.

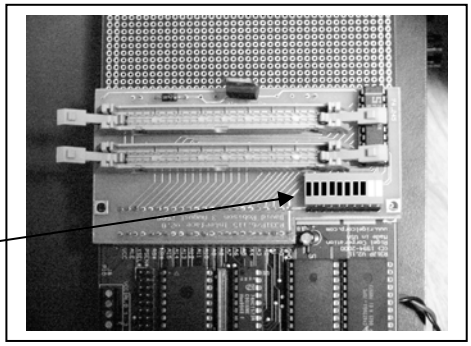
EXERCISE 4: Make a simple calculator

Let's make a simple calculator using the routines you developed in the previous exercises. Your calculator should allow the user to type in two numbers, each with exactly three digits, no more or less, using the PC keyboard. These numbers will be integers represented with a single byte, i.e., between 000 and 255. After the user has typed three digits, your program should automatically provide a carriage return and line feed to the next line. After two three-digit numbers have been entered, the user should type a + or – character on the PC keyboard, and the sum or difference, respectively, of the two three-digit numbers should be displayed on the LED bank. Do not worry about overflow or underflow, i.e., handling the result in some special way if it exceeds 255 or is less than zero. Do play with the program and observe what happens on the LED bank when overflow or underflow occurs. Take this as an opportunity to learn how the 8051 responds to overflow and underflow (this topic could show up on the quiz). Here's an example:

TeraTerm window:

```
MINMON>
*D                          (use transmit to send your
…….                         Calculator program)
002
001

                            (I hit "+" here, expecting
                            to get 3 in binary on the
                            LED bank )
```



You will write relatively substantial programs in this exercise. Make sure to use good coding practice, e.g., use subroutines, comments, and clear thinking to make the flow of your programs and algorithms very clear. We will ask you to explain during checkoff how commands like `LCALL` work with the stack.

Please do the following:

- Possibly helpful tips: The ASCII code for a number # is 3#h in hex, e.g., the ASCII code for the digit 9 is 57 in decimal or 39 in hex. Notice that adding d0h to the hex ASCII code for a digit # produces the result 0#h in hex. This trick can be used to directly convert ASCII codes for single digits back to the actual single digits.
- Write and test your simple calculator program. Notice that the program is intended to work a little like an HP calculator, i.e., the user "pushes" two three-digit numbers on to a stack, then enters an operation (addition or subtraction). This operation causes the two numbers to be popped off of the stack, added, and the results displayed automatically on the LED bank. The user does not have to press an "equals" sign to see a result. Make sure that your program uses the stack of the microcontroller, i.e., makes use of the `push` and `pop` instructions. Make sure that your program does not require a "reset" of the R31JP after each calculation. Your program should keep producing results as long as the user types arguments.
- Find out what happens when you overflow or underflow your program. For example, try adding the numbers `250` and `010`. Can you explain these results? How is the carry bit in the microcontroller involved in these calculations?

EXERCISE 5: Modify the calculator.

Let's make some changes to the calculator:

- Modify your calculator program so that, in addition to displaying the results on the LED bank, the program also displays the sum or difference on the PC screen. Here's a sample window:

```
MINMON>                 (use transmit to send your
*D                      Calculator program)
…….
002
001
                (I hit "+" here, expecting
003              to get 3 in binary on the LED bank and
                now to see 003 on the PC screen)
```

Even though this calculator program only adds or subtracts two numbers, you may find the `div ab` command useful for returning results to the PC screen. Notice that this command stores the remainder after the division in the `b` register.

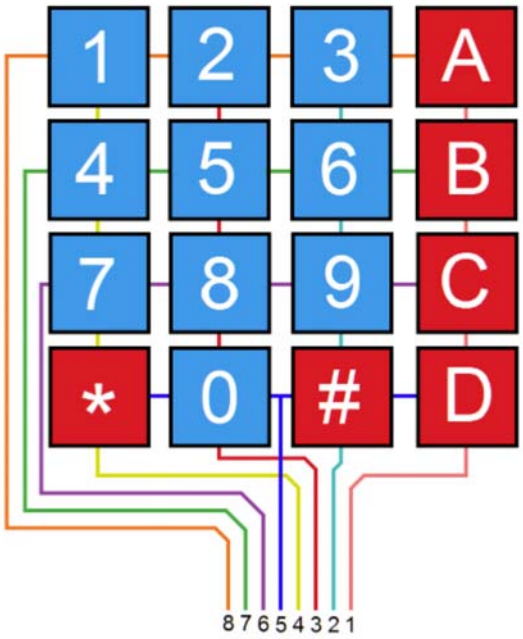
EXERCISE 6: Add a keypad to the calculator.

Modify your setup to include an official 6.115 keypad.

We will use this keypad throughout the term. Here's a picture of the pad:

You can find it in the "additional box" of parts that we issued to you when you checked out your kit.

The keypad is wired in a fairly simple and commonly used way. It has 16 keys arranged in a 4 by 4 matrix. When a key is depressed, a short circuit is created between one of the row pins and one of the column pins on the keypad connector. The four left most connections (labeled 8,7,6,5 in the schematic on the right) on the keypad connector correspond to the rows. The four rightmost (labeled 4,3,2,1) correspond to the four columns. Pressing a switch connects one row pin to one column pin. For example, pressing the "1" key creates a short between output pins 8 and 4.



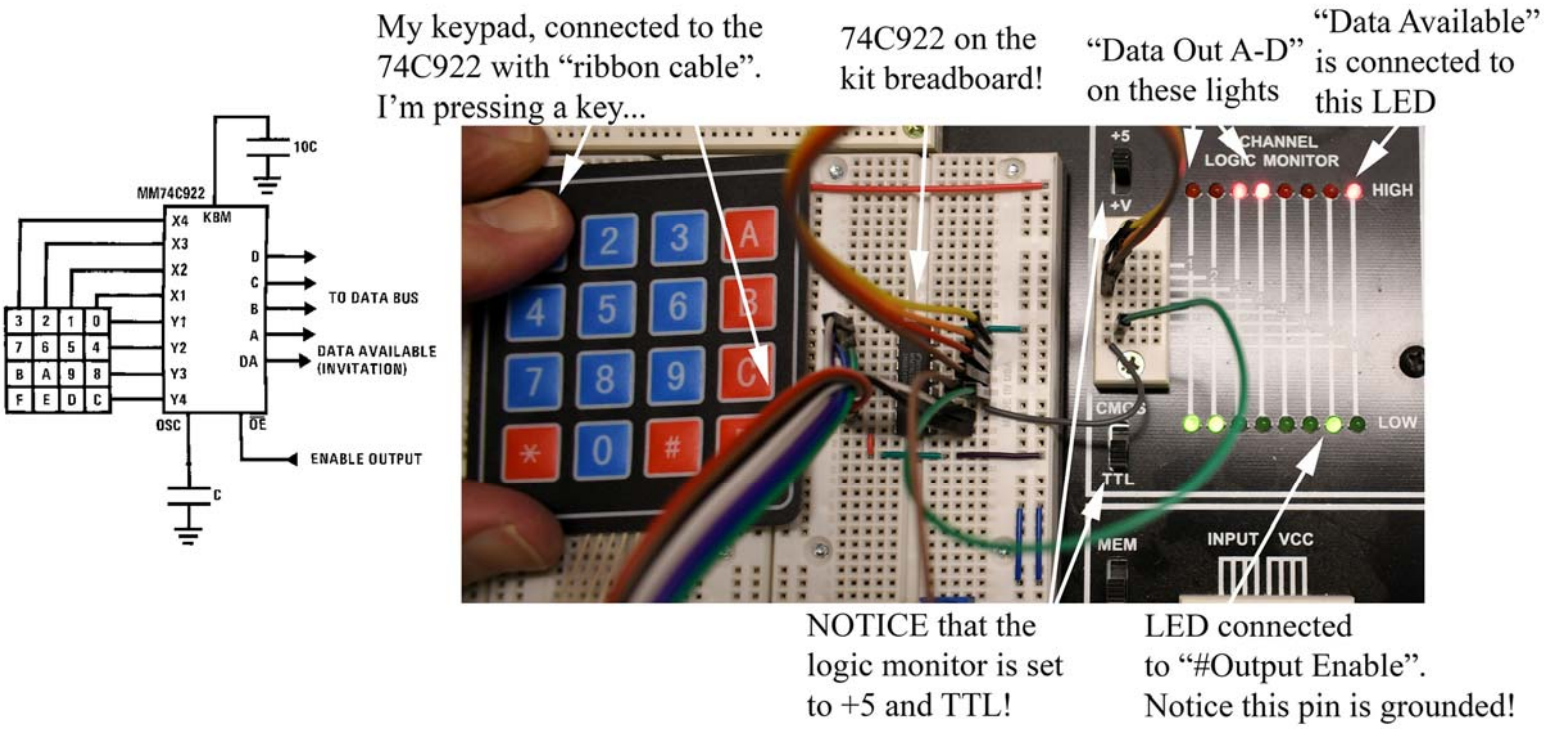
You can create a "ribbon cable" to connect your keypad to your breadboard by using the "pin-to-pin" jumpers in your kit. A pack of these jumpers is shown on the far right in the picture. The keypad has a connector with eight socket holes. Eight of the cable pins will fit in the eight sockets, as shown in the picture on the right. If you are careful, you can make an "eight pin" cable instead of ripping off eight separate pin-to-pin wires. Then, VERY carefully insert the eight pins (on one end of the cable) into the eight socket holes on the end of the keypad. **The keypad and connector are delicate! Be careful!** Also, be careful to AVOID twisting or crossing the wires in your cable. You want a straight cable that makes it easy to keep track of the pins and wire colors and connections to the keypad. (Sometimes it helps to squeeze the pins on the ribbon cable gently with forceps or pliers to make sure the pins are tapered carefully to enter the sockets.)

There are a number of ways we could choose to connect the keypad to the microcontroller. For this first laboratory, let's use a keypad controller chip, the 74C922, to connect the keypad to the R31JP board. ***READ the 74C922 spec sheet and understand how the chip works. (For example: Does this chip require external capacitors?)(The answer is YES – read and understand the spec sheet.) FOR 6.115(1), ONLY USE YOUR KEYPAD WITH THE 74C922.***

TEST your understanding of the keypad and the 74C922 before involving your R31JP. For example: You can use your multimeter on its "continuity" or "Ohms" setting with the multimeter leads connected to one row and one column wire on your ribbon cable. (Use alligator clips if you wish for convenient mechanical connection.) The multimeter should indicated "continuity" or "low ohms" when you press the appropriate key. Test several keys and multimeter connections; make sure you understand how the keypad is wired internally.

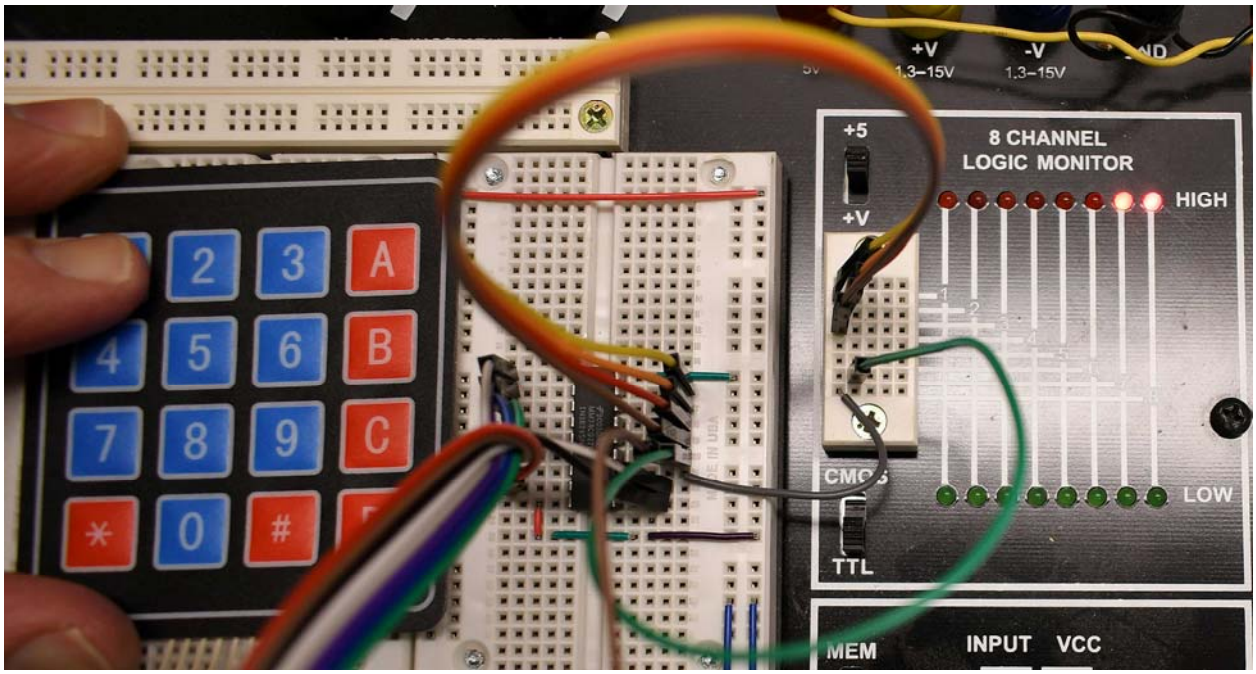
Next, you can connect your keypad to the appropriate row and column inputs of the 74C922. Configure the 74C922 so that at operates properly as described in the datasheet. Make sure you size your oscillator capacitor, KBM debounce ("keyboard mask") capacitor, bypass capacitor, and add other connections that are needed (power, ground, etc.). A partial schematic of the 74C922 circuit and the connections on the staff kit (no R31JP involved yet!) are shown below:

(NOTE: The staff kit is arranged slightly differently from the Blackbird; specifically, the logic monitor is in a different location. No worries – you have the same style of logic monitor available on your Blackbird. Please use your logic monitor.)



In the experiment on the staff kit illustrated above, the "Data Bus" output lines A-D are connected to the kit logic monitor lights, along with the "Data Available" output and active low "#OE" input of the 74C922. The #OE input is grounded in the experiment above, that is, the 74C922 will actively show any available data from a key press. When a key is pressed, the "Data Available" line goes high to let us know, and the code for the pressed key appears on the first four lights.

Notice also that when the #OE line is connected to +5 (high), the "Data Bus" A-D lines are "tri-state," i.e., in a high impedance mode, neither low nor high but more like an "open" connection, as shown in the figure on the next page. This "tri-state" or "high-Z" (high impedance) feature is useful when the 74C922 is sharing input lines to the microcontroller. We can actively turn of the output lines and make it appear, to the microcontroller, as if the 74C922 is simply "gone," and another device might then use the microcontroller lines.



Please use your kit and logic monitor to make sure that you understand the 74C922 before making any connections to the R31JP. Ask questions. When you are clear on how things work, ***then, move the 74C922 data lines to four pins on Port 1 of the R31JP. Connect the data available line and the output enable lines on the 74C922 to two free pins on R31JP Port 3.***

You can use the Port 3 lines to control the chip, and the Port 1 lines to read the data nibble indicating which key has been pressed. Note that by using the Port 3 control lines, you will be able to de-activate or "tri-state" the data lines connected to Port 1; this will allow you to use Port 1 for other peripheral chips.

Please do the following:

- ***Carefully (!) connect the 74C922 and keypad to Ports 1 and 3 as discussed. Make sure that you understand the chip and keypad so that you do not carelessly destroy anything. Use the FIVE volt supply to power your 74C922!***
- Write a simple program that allows you to verify that the keypad and 74C922 are working. When a key is pressed and released, display the data nibble from the 74C922 on the PC screen. Send a byte to the PC, with the most significant nibble set to a convenient value, and the least significant nibble set to the data from the 74C922. Write down a table that shows the data nibble provided by the 74C922 for each key. Your ASCII table may be helpful here. Note that the key pressed does not necessarily correspond to the nibble you get. That is, pressing the "9" key does not, unfortunately, give you 09h.
- Use the `db` instruction (see pages 93-96 in Yeralan) to create a data table called `keytab` that converts the data nibble from the 74C922 to a data byte that, at least for the digits on the keypad, allows a key press to be converted to a digit. That is, write an assembly routine that loads the accumulator with a byte that corresponds to the numerical value of a pressed key if that key is a digit.
- Create a calculator that, once again, computes the sum or difference of two three-digit numbers. Use the keypad to enter each of the three-digit numbers. Use the PC to show all digits and results. You may continue to use the PC to enter the + or – command, or, if you wish, you may define one of the

"free" keys on the keypad to be + and -. Be sure that your calculator does NOT require a reset after each calculation.

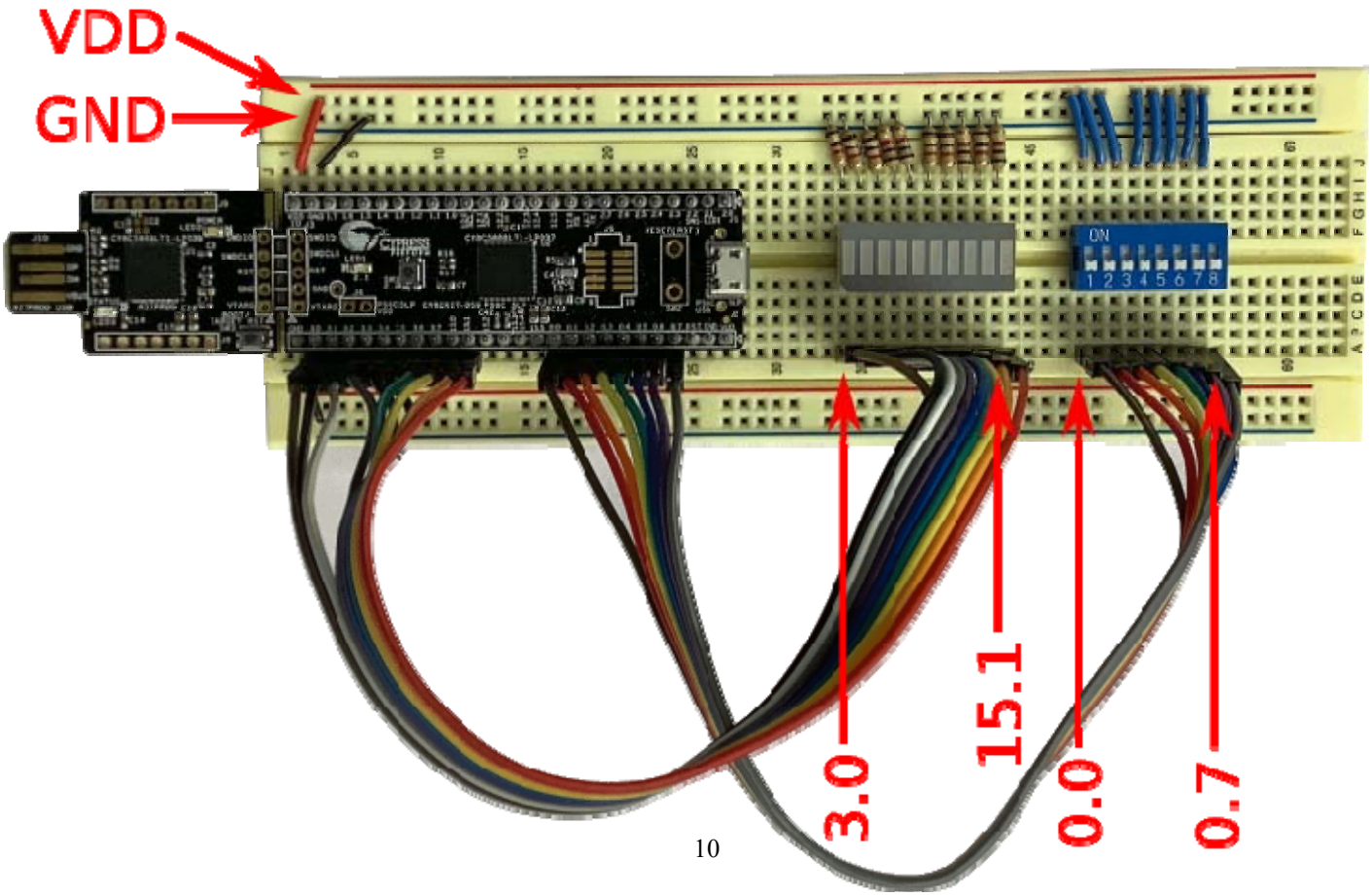
EXERCISE 6: Build your Kovid Konsole.

The Van Ess book (VE) has wonderful exercises that will both teach you more about digital logic and also let you experience the amazing flexibility of the PSoC hardware. In overview: we will work through a collection of lab exercises in VE. In each exercise, you will connect some PSoC hardware in Creator to make different arrangements of digital gates. To exercise the digital gate circuits that you build, you will need some switches to set some "inputs" high, and some LEDs with current limiting resistors to see or display the outputs of your digital circuit creations.

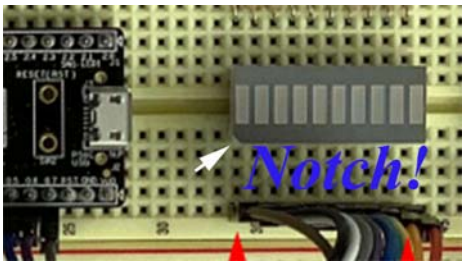
The input switches and output LED's are provided by a Kovid Konsole that you will build in this exercise.

The VE book uses a PSoC 4. We have given you a more powerful PSoC 5LP, which we will use for constructing the Kovid Konsole and for the associated PSoC exercises. So, particularly for the hardware pictures, e.g., Appendix A of VE, do not expect your hardware to "look" exactly the same. Please transpose to the appropriate pins for your PSoC 5LP Kovid Konsole. For example, VE refers to inputs on pins A0 – A7, which correspond to some pins on the VE PSoC 4. You too will have inputs A0–A7 connected to switches, but please locate the correct corresponding pins on your PSoC 5LP Kovid Konsole. The VE book also refers to output pins as "F#". You will have output pins that we will call F1 – F9. Again, please locate the appropriate pins on your Kovid Konsole to determine the correspondence for the VE exercises.

A picture of the Kovid Konsole you should please build is shown below:



Be SURE to use the current limiting 1K Ohm resistors, or you will damage the board and the LEDs. NOTICE that the LED bank has a "notch" on one side, closer to the STICK in the picture above. Make sure you orient your LED bank correctly. The resistors should be connected to the LED bank on the side with no "notch".



Some things to note:

- Please build your Kovid Konsole on a "spare" breadboard from your kit, NOT the main breadboard in your Blackbird. You have extra "strips" of breadboard in your box.
- Use your PSoC 5LP STICK.
- NOTE carefully the location of the +5 (VDD) and ground (GND) rails. For now, you will power your Kovid Konsole through the STICK's USB programming connector that hangs off the left edge of the breadboard in the picture; **add no other source of power**! You can use the USB extension cable in your kit to both program and power the Konsole from your PC. Note that the switch bank is connected to +5 and that the LED bank is connected to 1K resistors which are then connected to GND.
- USE your pin-to-pin cables to make ribbon cables that make your wiring job easy. The "silkscreen" text on the STICK indicates the PSoC pin numbers to you. We are using the PSoC pins labeled "3.0" to "15.1" to connect to the LED bank. The connections should be made beginning with P3.0 on the PSoC to the first LED in the bank (closest to the notch), and proceed sequentially through P15.1 on the PSoC Stick to the LED bank. We are using the PSoC pins 0.0 through 0.7, sequentially, to connect to the switch bank. A table of the pin out indicating the appropriate labels for the VE exercises is shown below:

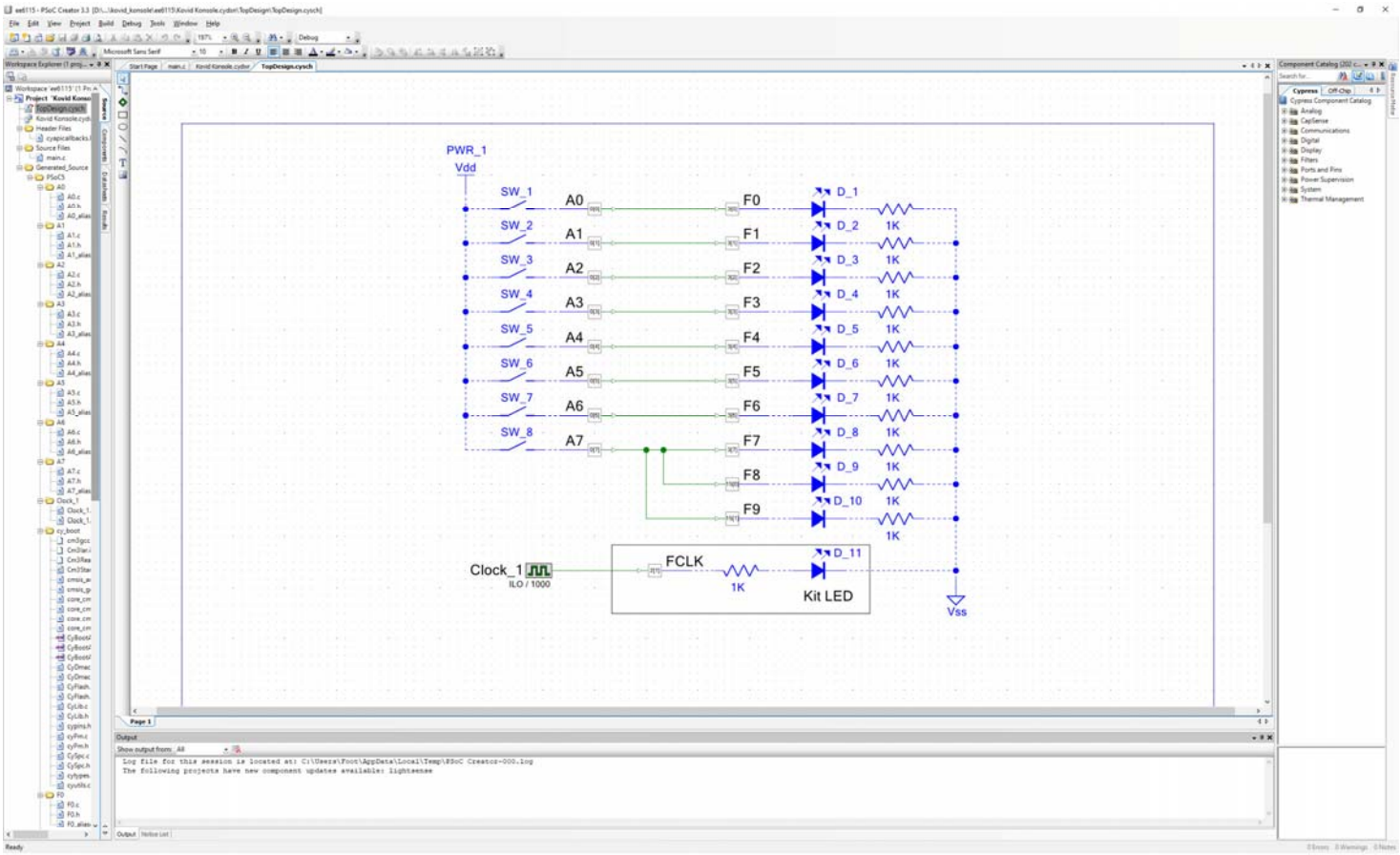
Inputs		Outputs	
Name	PSoC	PSoC	Name
A0	0[0]	3[0]	F0
A1	0[1]	3[1]	F1
A2	0[2]	3[2]	F2
A3	0[3]	3[3]	F3
A4	0[4]	3[4]	F4
A5	0[5]	3[5]	F5
A6	0[6]	3[6]	F6
A7	0[7]	3[7]	F7
		15[0]	F8
		15[1]	F9

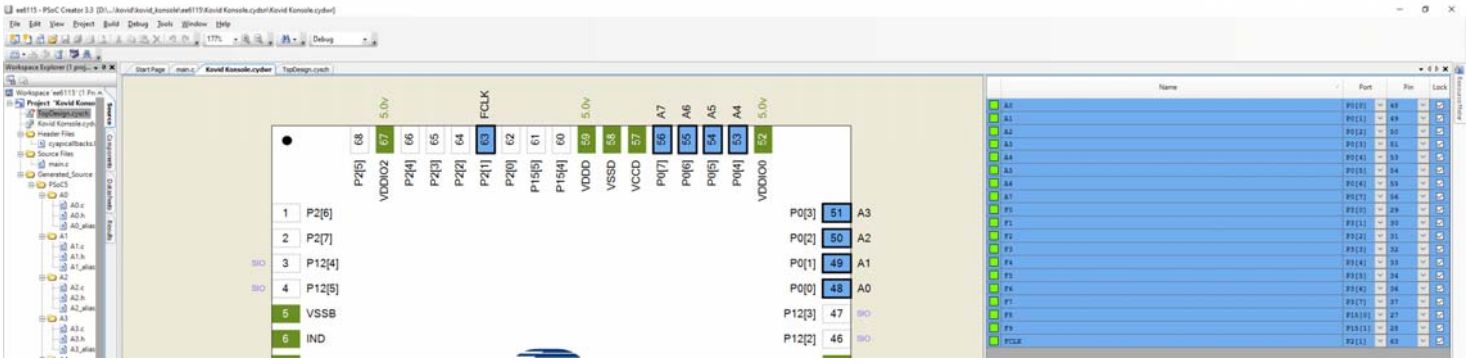
Please do the following:

- CAREFULLY wire up your Kovid Konsole.
- Include good pictures of your Konsole (like those above) in your lab report.

EXERCISE 7: Test your Konsole.

You now have a total of 8 switches on the Konsole, and 11 LEDs: 10 LEDs that you wired up using the LED bank, and one blue LED already available on the STICK. Set up a "blank" Creator project for a PSoC 5LP, CY8C5888LTI-LP097. Then, create a PSoC Creator project like this one shown below, with TopDesign shown first, and then Design wide resources (with pin assignments!):





Note that NO C programming is needed to use the simple PSoC hardware!

Note also that our staff "TopDesign" sheet shown above includes "blue" components for documentation and clarity. These components are NOT internal PSoC components (e.g., the switches SW_1 – SW_8 are the external switches on the Konsole). The internal PSoC project is simple, using 8 pins to receive switch inputs, 10 output pins to transmit the switch states to the LED bank, and an additional clock and output pin to create a blinking light out of the LED already available on the STICK. That is, your "TopDesign" sheet could look simpler than the one pictured above, and it would still provide the same functionality, with just the 19 pin connections and a clock, properly connected. The additional blue components used in the staff solution above are available in PSoC Creator (under the "Off-Chip" menus) to make the TopDesign schematic make more sense in the context of the Konsole.

Please do the following:

- Make your Creator project to test your Konsole. Use the Creator screen shots above as a guide for your work.
- Test your Konsole by trying different switch configurations. Active switches should create glowing LED's.
- Take screen shots of your Creator project. Document in your lab report.
- Take a picture of your Kovid Konsole with an interesting light and switch pattern that convincingly shows that your switches correspond to your light pattern. Document in your lab report.

(Mastery of keypads will be a critical skill in the future! Be the FIRST person to e-mail Professor Leeb with the exact title of the movie containing this scene above and receive a $10 Amazon gift certificate!)

ASCII Table: Fill in the missing characters using your "typewriter" program and LED light bank. PUT A COPY IN YOUR LAB NOTEBOOK!

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	00 0000 0000	01 0000 0001	02 0000 0010	03 0000 0011	04 0000 0100	05 0000 0101	06 0000 0110	07 0000 0111	08 0000 1000	09 0000 1001	10 0000 1010	11 0000 1011	12 0000 1100	13 0000 1101	14 0000 1110	15 0000 1111	
	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI	
0	□	⌜	⊥	⌟	↘	⊠	✓	卐	↶	→	≡	∨	⇓	←	⊗	⊙	8
	16 0001 0000	17 0001 0001	18 0001 0010	19 0001 0011	20 0001 0100	21 0001 0101	22 0001 0110	23 0001 0111	24 0001 1000	25 0001 1001	26 0001 1010	27 0001 1011	28 0001 1100	29 0001 1101	30 0001 1110	31 0001 1111	
	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	
1	⊟	◷	◶	◵	◴	✓	⎍	⊣	⧖	↓	ς	⊖	◰	◱	◳	◲	9
	32 0010 0000	33 0010 0001	34 0010 0010	35 0010 0011	36 0010 0100	37 0010 0101	38 0010 0110	39 0010 0111	40 0010 1000	41 0010 1001	42 0010 1010	43 0010 1011	44 0010 1100	45 0010 1101	46 0010 1110	47 0010 1111	
2	SP								(	)		+					A
	48 0011 0000	49 0011 0001	50 0011 0010	51 0011 0011	52 0011 0100	53 0011 0101	54 0011 0110	55 0011 0111	56 0011 1000	57 0011 1001	58 0011 1010	59 0011 1011	60 0011 1100	61 0011 1101	62 0011 1110	63 0011 1111	
3														=			B
	64 0100 0000	65 0100 0001	66 0100 0010	67 0100 0011	68 0100 0100	69 0100 0101	70 0100 0110	71 0100 0111	72 0100 1000	73 0100 1001	74 0100 1010	75 0100 1011	76 0100 1100	77 0100 1101	78 0100 1110	79 0100 1111	
4	@																C
	80 0101 0000	81 0101 0001	82 0101 0010	83 0101 0011	84 0101 0100	85 0101 0101	86 0101 0110	87 0101 0111	88 0101 1000	89 0101 1001	90 0101 1010	91 0101 1011	92 0101 1100	93 0101 1101	94 0101 1110	95 0101 1111	
5												[					D
	96 0110 0000	97 0110 0001	98 0110 0010	99 0110 0011	100 0110 0100	101 0110 0101	102 0110 0110	103 0110 0111	104 0110 1000	105 0110 1001	106 0110 1010	107 0110 1011	108 0110 1100	109 0110 1101	110 0110 1110	111 0110 1111	
6							f							m			E
	112 0111 0000	113 0111 0001	114 0111 0010	115 0111 0011	116 0111 0100	117 0111 0101	118 0111 0110	119 0111 0111	120 0111 1000	121 0111 1001	122 0111 1010	123 0111 1011	124 0111 1100	125 0111 1101	126 0111 1110	127 0111 1111	
7											z			}		DEL	F