

ELEC 3040/3050

Lab 5

Matrix Keypad Interface
Using Parallel I/O

Goals of this lab exercise

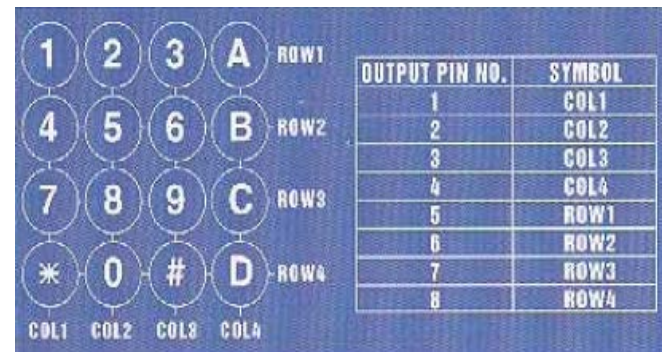
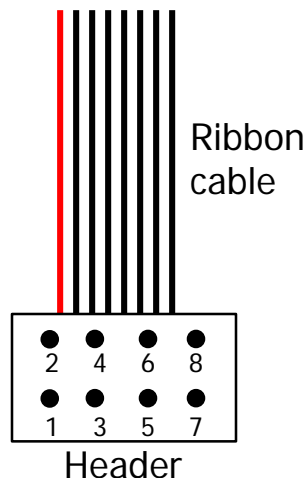
- Control a “real device” with the microcontroller
- Coordinate parallel I/O ports to control and access a device
- Implement interrupt-driven operation of a device
 - External device triggers operation(s)

Velleman 16-Key Matrix Keypad

- Used on phones, keyless entry systems, etc.
- Comprises a matrix of switches
 - no “active” circuit elements
- Accessed via 8 pins (*4 rows/4 columns*)
 - connected by a ribbon cable to a DIP header
 - insert carefully into breadboard



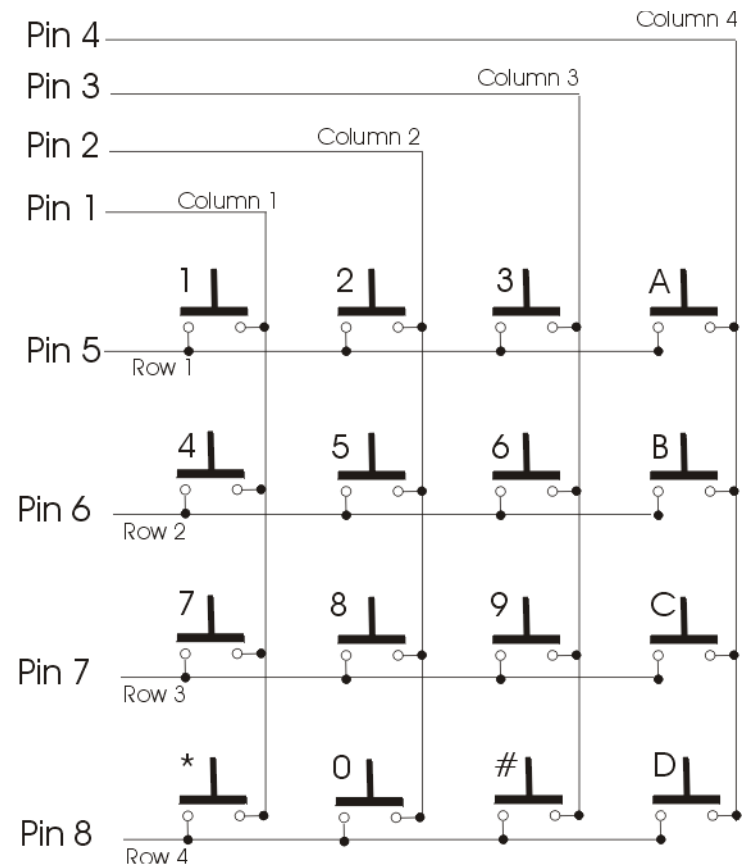
Pins: 1-2-3-4-5-6-7-8



Pin connections to rows/columns

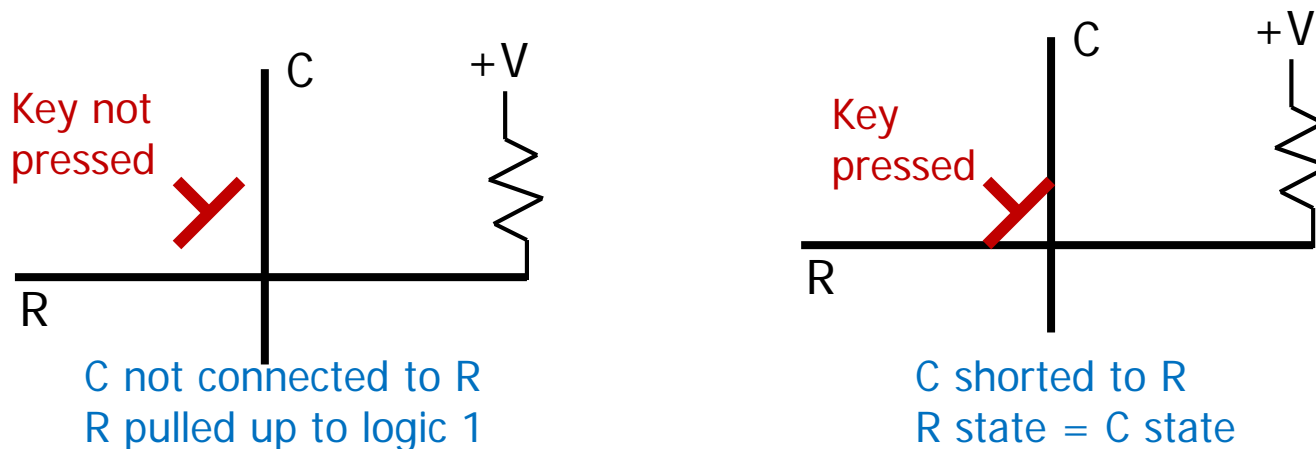
Matrix keypad circuit diagram

- 16 keys/contact pairs
 - 4 rows x 4 columns
 - One key at each row-column intersection
- Springs normally separate keys from contacts
- Pressing a key connects the contacts
(“short circuits” row-to-column)



“Scanning” the keypad

- Drive column wires with output pins
 - Drive a column wire low to “activate” it
- Read states of row wires via input pins
 - Use *pull-up resistors* to pull rows up to logic 1
 - If no row-column shorts, all rows pulled high
 - Row R low only if shorted to column C that is driven low

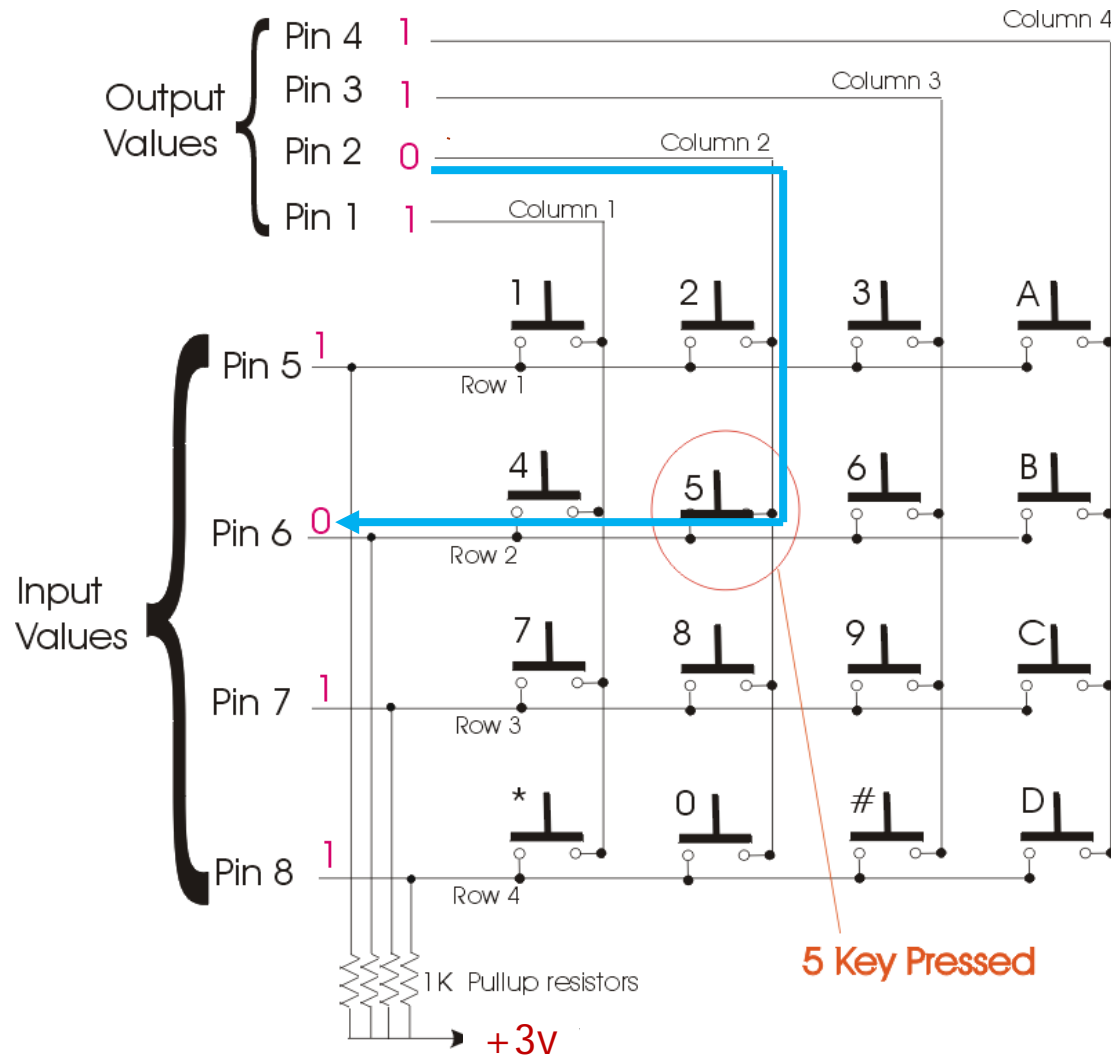


Scan algorithm

1. Drive one column C low and other columns high
2. Read and test the states of the rows
 - If row R is low, it is shorted to column C
 - If row R is high, then *either*:
 - R is not shorted to any column wire & remains pulled high
 - *or*, R is shorted to a column wire that is being driven high
3. Repeat steps 1 & 2, but with a different column wire driven low (and others high)
 - Key press detected if a row is detected in a low state
 - Key position is intersection of that row and the column being driven low

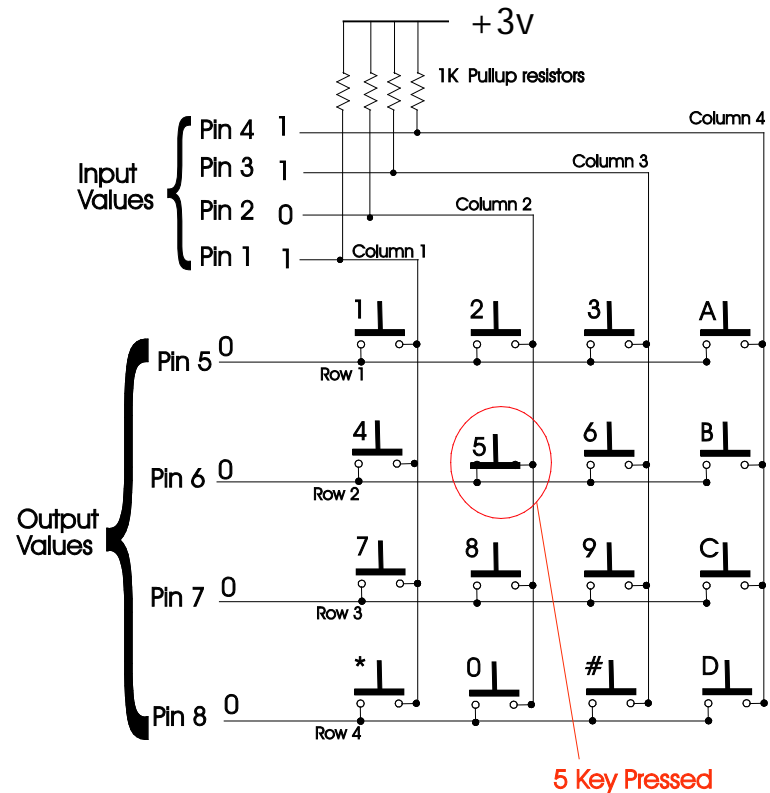
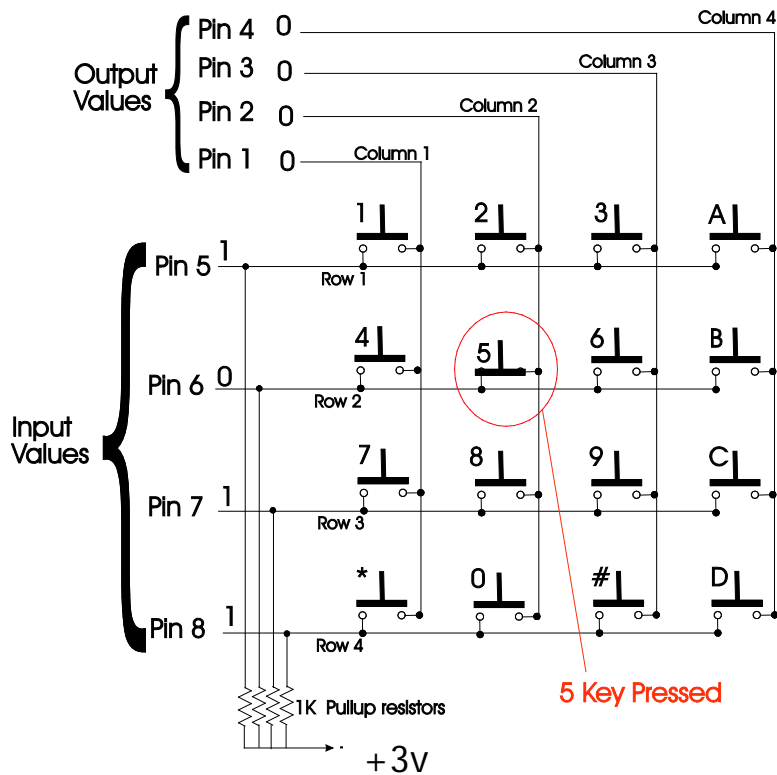
(example on next slide)

Example *(C-2 driven low, R-2 detected low)*



Alternate (non-scan) method

- (1) Write to columns (out) and read rows (in)
- (2) Change port directions (via MODER)
- (3) Write rows (out) and read columns (in)



Timing issue

- There is a **short time delay** from the time a pattern is written to an output port to the appearance of that pattern on the external pins.
- After writing a pattern to an output port (to drive column lines), **insert a short program delay** (a few “dummy instructions”) before reading the input port (to test the row lines.)

Example:

```
write to output port;  
for (k = 0; k < 4; k++); //do-nothing loops for delay  
read input port;
```

GPIO pin electronics - pull-up/pull-down control

- Use pull-up/down device to create a default logic state on a pin
 - For inputs that are not always driven (or driven by open-collector/drain ckt)
 - Often pull unused input pins high or low logic level to prevent CMOS latch-up
- STM32L1xx GPIO has **internal** pull-up/pull-down devices for each pin
 - Activate via register **GPIO_n->PUPDR**

32-bit register/2 bits per pin:

31	30	7	6	5	4	3	2	1	0
Px15		Px3		Px2		Px1		Px0	

00: No pull-up or pull-down (reset state for all but PA[15:13], PB[4])

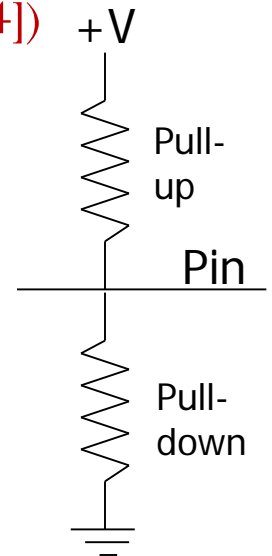
01: Activate pull-up

10: Activate pull-down

Example: Activate pull-up resistor on pin PA3

```
GPIOA->PUPDR &= ~0x000000C0; //clear bits 7-6 for PA3
```

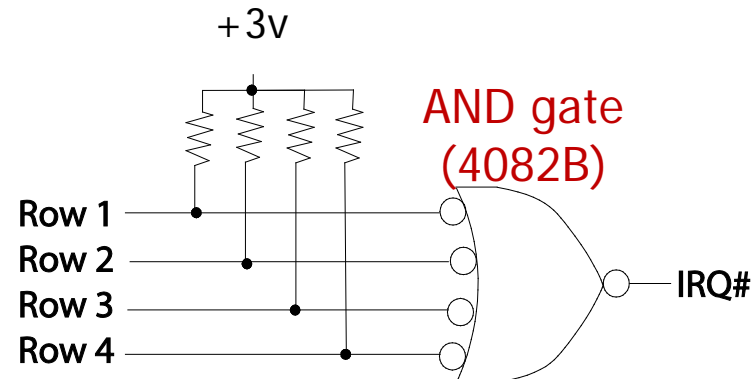
```
GPIOA->PUPDR |= 0x00000040; //set bits 7-6 to 01 for PA3 pull-up
```



Keypad interrupt signal - hardware

- Generate an interrupt signal when any key is pressed
 - Drive all columns low
 - Logical “OR” active-low rows
 - Any low row triggers IRQ#
 - AND gate:

$$\overline{\overline{A} + \overline{\overline{B}} + \overline{\overline{C}} + \overline{\overline{D}}} = ABCD$$



- Connect IRQ# to a GPIO pin, configured as EXTI interrupt
 - Configure EXTI as falling-edge triggered and enable it in EXTI and NVIC
 - Falling edge sets “pending” bits in EXTI and NVIC to trigger an interrupt
 - Interrupt handler must clear the pending bit in EXTI
 - Pending bit in NVIC is automatically cleared when the interrupt handler is executed, **but may set again if switch bouncing occurs!**

See “bouncing” on next slide

Dealing with “key bounce”

- Mechanical switches often exhibit **bouncing**
 - Multiple state changes during switch closure/opening
 - Due to electrical arcing or mechanical settling of contacts
- (Assume IRQ falling-edge-triggered)*



- Multiple state changes might trigger **multiple interrupts**, when only one interrupt is desired/expected *(the above could trigger 5 interrupts)*
- **Debouncing** may be required to ensure a **single response** to a switch activation

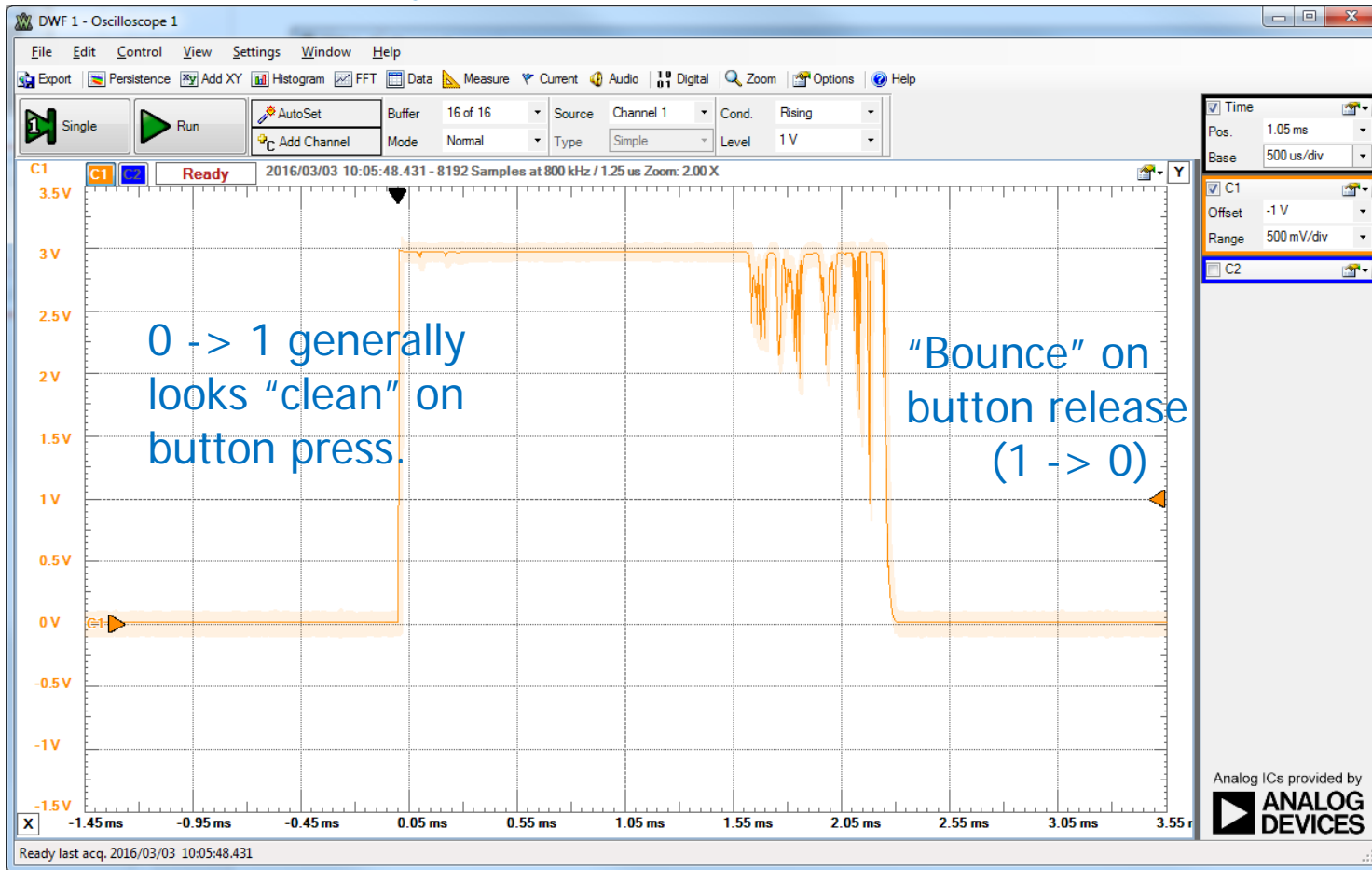
Example: Interrupt triggered by initial state change. Delay until bouncing finished, and then clear “pending registers” in both EXTI and NVIC.

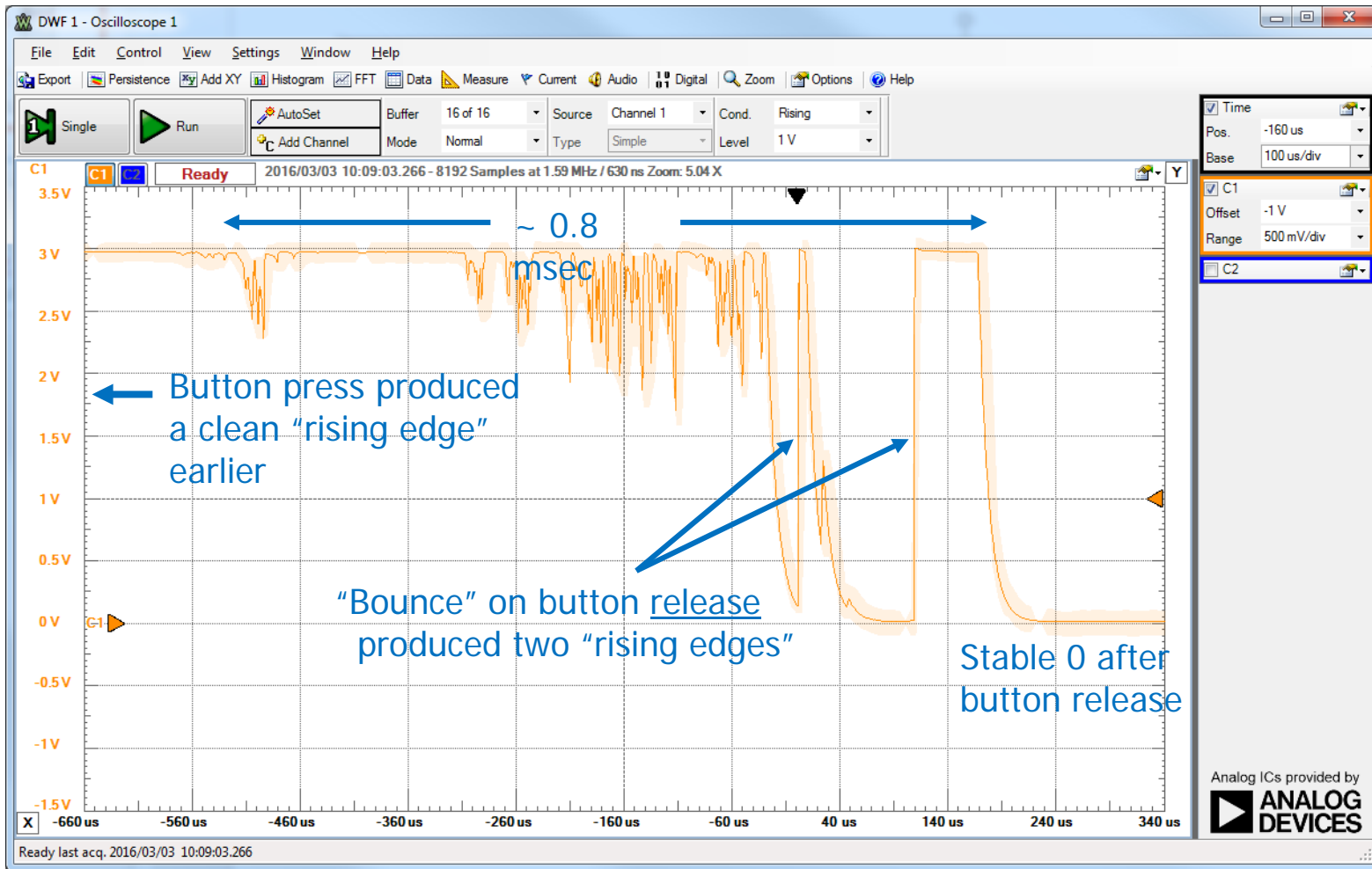
```
EXTIx_IRQHandler() {  
    - do required operations for this interrupt  
    - delay at least  $T_{\text{bounce}}$   
    - clear EXTI and NVIC “pending” bits for this interrupt  
} // no more pending interrupts after exiting the handler
```

Possible debugging test:

- Increment a variable in the interrupt handler.
- Should increment only once per button press.

Discovery Board User Button (PA0) – “switch bounce”





Hardware design

- Insert keypad into breadboard and connect microcontroller GPIO pins to “devices” as shown below.
- In software - activate internal pull-up resistors on row lines

GPIO Pins	Connected Devices
PB3-PB0	Keypad rows 4-1 (inputs)
PB7-PB4	Keypad columns 4-1 (outputs)
PC3-PC0	LEDs (outputs)
PA1	IRQ#
Other ports	Additional LEDs for debug

Also: connect PB7-PB0 and PA1 to EEBOARD DIO pins and use logic analyzer/oscilloscope to help debug connections and the scanning algorithm.

Software design

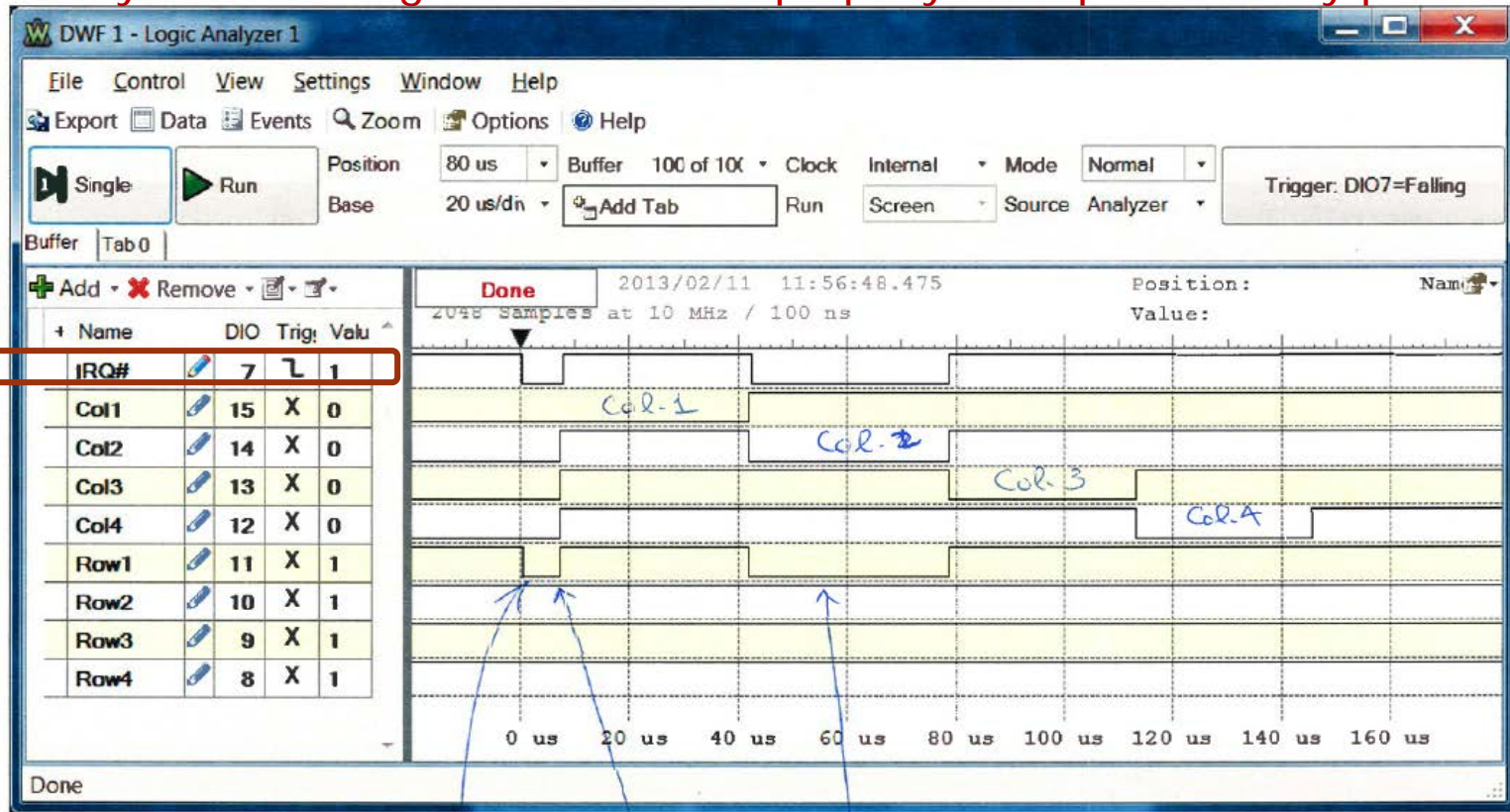
- Review how to read/write I/O ports, set/clear/test bits, and set up a GPIO pin to interrupt the CPU
- Main program
 - Perform all initialization
 - Run in a continuous loop, incrementing a one-digit decimal count once per second, displayed on 4 LEDs
 - Don't display the count for 5 seconds following a key press, but do continue counting while the key number is displayed
 - Resume displaying the count after 5 seconds has elapsed
- Keypad interrupt handler (executed when key pressed)
 - Determine which key was pressed
 - Display the key number on the 4 LEDs
 - Set a global variable so the main program will know to leave the key# displayed for 5 seconds
 - Perform any debouncing and clear pending flags
- Notes:
 - After reading inputs from a port – mask all but the row bits
 - Consider a “scan loop” rather than 16 “if” statements for detecting keys (repeated operations)

Debug suggestions

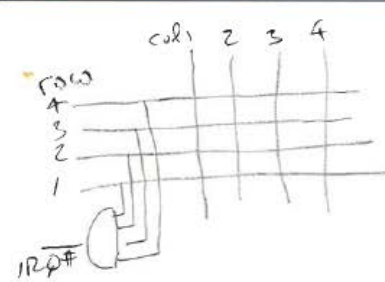
- Observe one or more global variables in a **Watch window**
 - Increment a variable in the ISR to count the number of times the ISR is executed
 - Indicates interrupt detected and ISR entered
 - Detects multiple interrupts on one key press (due to “key bouncing”)
 - Set a variable to the pressed key number
 - Set a variable to values that represent steps of an algorithm
- A switch can be connected to PA1 instead of the keypad to manually trigger interrupts to test the ISR
- The ISR can write some unique pattern to LED(s) to indicate that it was entered
- Use the oscilloscope to investigate “key bounce” (trigger oscilloscope on first interrupt signal).
 - Is bouncing observed?
 - How long does it last?

Debugging with a logic analyzer

Verify that scan algorithm executes properly in response to key press.



Trigger



IRQ# activates

ISR begins the scan algorithm.

Row 3 / Col. 2 Key press. (note that IRQ# activates here)