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
  - Interrupt-driven: external device triggers data transfer
  - Program controlled: software controls timing of data transfers
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
- Spring normally holds key away from contacts
- Pressing a key connects the contacts (“short circuits” row-to-column)
“Scanning” the keypad

- Drive column wires with output port
  - Drive a column wire low to “activate” it
- Read states of row wires via input port
  - 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

![Diagram](https://via.placeholder.com/150)

Key not pressed

- C not connected to R
- R pulled up to logic 1

Key pressed

- C shorted to R
- R state = C state
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 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 and testing the keypad 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 `GPIOOn->PUPDR`

<table>
<thead>
<tr>
<th>32-bit register / 2 bits per pin:</th>
<th>31 30 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Px15 Px3 Px2 Px1 Pxo</td>
<td><strong>Pull-up</strong></td>
</tr>
<tr>
<td>00: No pull-up or pull-down (reset state for all but PA[15:13], PB[4])</td>
<td></td>
</tr>
<tr>
<td>01: Activate pull-up</td>
<td></td>
</tr>
<tr>
<td>10: Activate pull-down</td>
<td></td>
</tr>
</tbody>
</table>

**Example:** Activate pull-up resistor on pin PA3

```c
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:
    \[
    \overline{A} + \overline{B} + \overline{C} + \overline{D} = A\overline{B}C\overline{D}
    \]

- Connect IRQ# to a GPIO pin, configured as EXTIIn interrupt
  - Configure EXTIIn 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 may trigger multiple interrupts, when only one interrupt is desired/expected (the above could trigger 3 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_{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.

Observe/measure $T_{bounce}$ on oscilloscope
Discovery Board User Button (PA0) – “switch bounce”

0 -> 1 generally looks “clean” on button press.

“Bounce” on button release (1 -> 0)
"Bounce" on button release produced two "rising edges".

Button press produced a clean "rising edge" earlier.

Stable 0 after button release.

~ 0.8 msec
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

<table>
<thead>
<tr>
<th>GPIO Pins</th>
<th>Connected Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB3-PB0</td>
<td>Keypad rows 4-1 (inputs)</td>
</tr>
<tr>
<td>PB7-PB4</td>
<td>Keypad columns 4-1 (outputs)</td>
</tr>
<tr>
<td>PC3-PC0</td>
<td>LEDs (outputs)</td>
</tr>
<tr>
<td>PA1</td>
<td>IRQ#</td>
</tr>
<tr>
<td>Other ports</td>
<td>Additional LEDs for debug</td>
</tr>
</tbody>
</table>

Also - connect PB7-PB0 and PA1 to EEBOARD DIO pins and use the 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.