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
  - Pull row wires up to logic 1 with *pull-up resistors*
  - If no row-column shorts, all row wires are pulled high
  - Row wire R can only be low if it is shorted to a column wire C that is being driven low

![Diagram showing key states](image-url)
Scan algorithm

1. Drive column wire C low, and other columns high
2. Read and test the states of the row wires
   • 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)
   • A key press is detected if a row line is detected in the low state
   • Key position is intersection of that row and the column being driven low
Example  
(C-2 driven low, R-2 detected low)

Output Values

<table>
<thead>
<tr>
<th>Pin 1</th>
<th>Pin 2</th>
<th>Pin 3</th>
<th>Pin 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Input Values

<table>
<thead>
<tr>
<th>Pin 5</th>
<th>Pin 6</th>
<th>Pin 7</th>
<th>Pin 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Row 1: 1, 4, 5, 8
Row 2: 2, 3, 5, 6
Row 3: 7, 5, 9
Row 4: *0, #

1K Pullup resistors

+3v

5 Key Pressed
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:

```c
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

- May need 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`

  32-bit register / 2 bits per pin:

  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

```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 a key is pressed
  - Logical “OR” of active-low rows
  - All column lines driven low
  - Any low row triggers IRQ#
  - AND gate: \[ \overline{A} + \overline{B} + \overline{C} + \overline{D} = ABCD \]

- Connect IRQ# to a GPIO pin, configured as EXTIn interrupt
  - Configure EXTIn 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 the 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.

```c
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.
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 that the interrupt was detected and the 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 one or more LEDs 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.
Alternate (non-scan) method

1. Write columns (out) and read rows (in)
2. Change port directions
3. Write rows (out) and read columns (in)