AVR243: Matrix Keyboard Decoder

Features
- 64-key Push-button Keyboard in 8 x 8 Matrix
- No External Components Required
- Wakes Up from Sleep Mode on Keypress
- Easily Implemented into Other Applications
- Low Power Consumption
- Software Contact Bounce Elimination
- Support for Alternate Function Keys, Easily Removable to Reduce Code Size
- Suitable for Any AVR® with a Minimum of 17 IO-lines and Pin Change Interrupt (Currently Only ATmega162 and ATmega169)
- Can Be Modified for Other Devices, Using One Common Interrupt, See Application Note “AVR240: 4 x 4 Keypad – Wake-up on Keypress”

Introduction
This application note describes a software driver interfacing an 8 x 8 keyboard. The application is designed for low power battery operation. The AVR spends most of its time in Power-down sleep mode, waking up when a key is pressed. The keyboard is scanned, scancodes are processed and the AVR finally reenters sleep mode.

The application also supports user-defined alternation keys to implement Caps Lock, Ctrl-, Shift- and Alt-like functionality. A test application implements a 4 x 4 keyboard with one digit and three letters on each key. Alternation keys choose the keys’ functions.

Figure 1. Application
The application is suitable for all applications using matrix keyboards, e.g., remote controls, mobile phones and alarm and access systems. It is also easily upgradable using the AVR’s In-System Programming capability or the Self-programming feature of the ATmega devices. (See application notes “AVR910: In-System Programming” and “AVR109: Self-programming”). An example on using the Self-programming feature is a general purpose re-programmable remote control.

The implementation described in this application note is using the ATmega162 device as target. The code can, however, be used with the ATmega169 device with minor modifications.

Theory of Operation

The keyboard matrix is organized in 8 x 8 pushbuttons connected with row and column lines as shown in Figure 2. A keypress connects the key’s row and column lines. When pressing the top left key, the leftmost column line is connected to the topmost row line.

Matrix keyboards can be scanned in several ways. When only single keys are pressed, a quick method is to first select (drive low) all row lines and read the column result. Then all column lines are selected, and the row result is read. Returned column and row is combined into a unique scancode for the specific key pressed. This method is used in this application note.

When simultaneous keypress capability is required, the method above cannot be used. The rows must be scanned separately. The row lines must be selected (driven low) sequentially, reading the column result for each row, thus getting all pressed keys. One limitation is when keys are pressed in such a pattern that unwanted interconnections appear. In Figure 3, the three highlighted keys pressed connect the row and column lines in such a way that the X-marked key appears to be pressed too. Therefore, this “ghost” key represents an error state.
To detect a keypress and wake-up from sleep mode, the pin change interrupt on the AVR is used. Prior to entering sleep mode, all rows are selected (driven low), thus driving a column line low on any keypress in any row. (The result of a key will cause an interrupt, but will decode as “No key pressed”.)

Due to the use of the pin change interrupt, only one key event is caught per key press. Automatically repeating keys, by interrupt, are not possible with this implementation. Repetition must be handled separately by the main application, by manually repeating the key’s associated action until all keys are released.

**Alternation Keys**

Many keyboard interfaces use secondary key functions. This can be implemented in several ways. One common method is to dedicate a number of keys to be alternation keys. When pressed simultaneously with ordinary keys, secondary scan codes are generated. It is also common to use sequential combinations, where an alternation key pressed prior to an ordinary key generates secondary codes. This eliminates the need to handle simultaneous key presses.

The alternation key(s) only applies to the first ordinary keypress after the alternation key(s) has been activated. This type of alternation key is in this document referred to as a “one-shot” alternation key.

It is also possible to implement the alternation keys as a toggle function, where the first press on the alternation key enables the secondary mode, and the next disables it again. The alternation applies to all keys pressed subsequently and is only terminated by second press on the alternation key. This is used on ordinary PC keyboards’ Caps Lock keys. This application uses a combination of both one-shot and lock-based alternation keys.

By using sequential keys rather than simultaneous keys, the problem with ghost keys is avoided. The application simply ignores all but the top left of the keys pressed.
Contact Bounce

When a keyboard button is pressed, the contact bounces for some time before settling in a steady on-state as illustrated in Figure 4. This must be handled in a way that won’t generate multiple keypresses. One common way to do this is to wait a short while from the keypress detection, letting the contact settle, and then read the actual state. This also eliminates errors due to noise spikes on the lines. Alternatively, a hardware implemented bounce eliminator or software digital filter can be applied. This application uses the first method, simply waiting for the contact to settle, since it is the most inexpensive and uses least power.

Figure 4. Contact Bounce

Implementation

The following implementation uses the ATmega162 device. Guidelines on porting to the ATmega169 device are listed on page 10.

The keyboard is connected to the AVR using two 8-bit IO ports. One port (Port D) is configured as output and is connected to the row selection lines. One port (Port C) is configured as input and is connected to the column return lines. See Figure 2 for details. When scanning the keyboard matrix, the port currently used for output must be driven low, while the input port must be internally pulled high, waiting to be driven low due to a keypress.

This application’s test function also uses Pin 1 of Port E as a serial output for transmitting the keypress information. See application note “AVR306: Using the AVR UART in C” for details using the UART.

An ordinary keypress generates a scancode in the range 0 - 63 (8 output lines * 8 input lines) even if not all rows and columns are used. Pressing an alternation key also generates ordinary scancodes, but in addition the alternation state flags are updated accordingly. The state flags are contained in a global variable.

Three of the alternation keys are configured as lock keys, enabling secondary mode on all following keys until pressed a second time. The lower four is one-shot, changing only the next key’s function before the Flag is cleared automatically. See Figure 5 for more details. When pressed, alternation keys also returns an ordinary scancode, allowing the application to handle alternation keys as ordinary keys.

Another global variable is used to pass the scancode together with the alternation Flags to the application. The six lower bits are used to indicate the code (0 - 63), while the MSB (bit 15) indicates the update status. The keyboard driver sets the bit when a key is
pressed. The application polls this bit and clears it when it has read the scancode. The global byte and word are illustrated in Figure 5.

**Figure 5. Extended Scancode**

![Extended Scancode Diagram](image)

The keyboard software is implemented as an interrupt-controlled driver. The main application enters sleep mode when waiting for keypresses. The keyboard driver wakes up the AVR when a key is pressed, gets the scancode and updates the global bytes. The main application has the responsibility of reentering sleep mode, since a SLEEP instruction inside an interrupt would halt the CPU.

**Initialization and Main Loop**

The driver initialization and main loop routine is shown in Figure 6. The alternation Flags and global variables are cleared and the ports are initialized according to the previous description. Power-down sleep mode is selected, allowing the application to enter sleep mode when no processing is required.

**Figure 6. Initialization**

![Initialization Diagram](image)
Keyboard Decoder Main Loop

The driver’s main loop is executed on pin change interrupt. It first selects Idle sleep mode, allowing the AVR to wake-up on Timer Overflow, since a Timer Overflow Interrupt will not wake up the AVR from Power-down sleep mode. The timer is then set up to wait a short while (5 ms typical) for the contacts to settle. During this delay, the main application regains control and may enter sleep mode. When the delay is finished, the keypress processing function is called. This completes the retrieval of a keypress. Finally, all row lines are driven low and Power-down sleep mode is selected, allowing the main application to enter sleep mode and wake up on next keypress. The flow chart for this loop is shown in Figure 7.

Figure 7. Keyboard Decoder Main Loop
Keyboard Scanning (Process Keypress)

The keyboard scanning process is illustrated in Figure 8. First the column result is scanned. The three lower bits (column part) of the scancode is incremented until the low column line is found. Then, the port directions are inverted and the row result is scanned. The row part of the scancode is incremented until the low row bit is found. Finally, the scancode processing function is called.

**Figure 8.** Keyboard Scanning

Check Column Result

Process Keypress

Test Column Result

Get Column Position

Any Columns Low?

Yes

Initialize Temporary Scancode

RETURN

No

Check Row Result

LSB of Column Result Low?

Yes

Invert Port Directions

LSB of Row Result Low?

Yes

Process scancode

No

Set Original Port Directions

RETURN

No

Shift Column Result Right 1 Bit

Drive All Column Lines Low

Increment Column Part of Scancode

Test Row Result

Any Rows Low?

Yes

Set Original Port Directions

RETURN

No

Shift Row Result Right 1 Bit

Increment Row Part of Scancode

RETURN
The scancode processing function first copies the generated scancode to the global variable. Then the scancode is compared to the predefined alternation key codes, updating alternation Flags accordingly. The flags are copied to the global variable. If the key wasn’t an alternation key, the one-shot Flags are cleared. Finally, the update Flag is set, indicating a new keypress has been received. The procedure is shown in Figure 9.

**Figure 9. Scancode Processing**

The Test Application – myCellPhone

The source includes a simple test application that implements a cellphone-like keyboard. Resulting characters are transmitted using the USART. Using a 4 x 4 keypad, all digits and letters are available using four alternation keys. The digit keys alone will transmit digits. Three alternation keys are used to choose between the three letters on each digit key, and finally a Caps-Lock key chooses between lowercase and uppercase.

Conversion tables are used to convert scancodes to characters, based on the current alternation state. The scancode is used as an index into the tables. The scancode is also a direct representation of the key’s position on an 8 x 8 keypad. The tables must therefore have eight entries per row, even if the keypad only has four keys per row. If the test application is modified to utilize larger keypads, it is, therefore, most size-efficient to add columns and not rows to the keypad.
Code Size and Timings

The code size for the keyboard matrix interface functions is shown in Table 1.

Table 1. Keyboard Matrix Code Size

<table>
<thead>
<tr>
<th>Function</th>
<th>Code Size (Words)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_init</td>
<td>25</td>
<td>Keyboard interface initialization</td>
</tr>
<tr>
<td>key_stop</td>
<td>7</td>
<td>Temporary stop the driver</td>
</tr>
<tr>
<td>key_get</td>
<td>15</td>
<td>Wait for a keypress</td>
</tr>
<tr>
<td>key_processAltKeys</td>
<td>88</td>
<td>Process the alternation keys and update flags accordingly</td>
</tr>
<tr>
<td>pinChangeISR</td>
<td>24</td>
<td>Interrupt handler for pin change interrupt</td>
</tr>
<tr>
<td>Timer0OVFISR</td>
<td>107</td>
<td>Interrupt handler for timer overflow interrupt</td>
</tr>
<tr>
<td>Total</td>
<td>266</td>
<td></td>
</tr>
</tbody>
</table>

When running at 8 MHz on the ATmega162, the following execution timings in Table 2 were measured. It shows the sequence from wake-up on the first detected keypress, to re-entering sleep mode after keypress processing.

Table 2. Code Timings

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin change wake-up -&gt; Init timer -&gt; Enter idle sleep</td>
<td>40 µs</td>
</tr>
<tr>
<td>Idle sleep -&gt; Timer overflow</td>
<td>65.4 ms</td>
</tr>
<tr>
<td>Timer wake-up -&gt; Keyboard scanning -&gt; Enter Power-down sleep</td>
<td>351 µs</td>
</tr>
<tr>
<td>Total time not in Power-down sleep mode</td>
<td>65.7 ms</td>
</tr>
</tbody>
</table>

When not in Power-down sleep mode the keyboard interface spends most of its time waiting in Idle sleep mode. The device is in Active mode only 0.3 ms, which means 0.5%. The relative time spend in the different modes is listed in Table 3.

Table 3. Time Spend in Sleep/Active Mode

<table>
<thead>
<tr>
<th>Mode of Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>0.5 % (15 mA)</td>
</tr>
<tr>
<td>Idle sleep</td>
<td>99.5 % (7 mA)</td>
</tr>
<tr>
<td>Power-down</td>
<td>When waiting for a keypress (&lt; 1 µA)</td>
</tr>
</tbody>
</table>

Consider that a key is pressed every 10 minutes. This would give an average current consumption of less than 2 µA.
Porting Considerations

The only difference from ATmega162 to ATmega169 affecting this application is the port connected to the column lines of the matrix keyboard. ATmega162 uses Port C, while Port B or E must be used on ATmega169. This is due to the Pin Change Interrupt pin configuration. ATmega162 has Pin Change Interrupt capabilities on Port A and C, while ATmega169 uses Port B and E for these interrupts. The application can easily be modified to use the other port capable of handling Pin Change Interrupts, if the default port’s alternate capabilities are needed.

Also note that ATmega169 uses the SMCR Register instead of MCUCR for sleep mode selection.

Porting this application to devices without Pin Change Interrupts requires the use of external components and different interrupt usage. This is covered in application note “AVR240: 4 x 4 Keypad – Wake-up on Keypress”. The rest of the scanning functionality in this application needs no changes.