AN2034

PSoC® 1 – Reading Matrix and Common Bus Keypads

Author: Dave Van Ess, Rajiv Badiger

Associated Project: Yes

Associated Part Family: CY8C27xxx, CY8C29xxx, CY8C21x23, CY8C21x34,CY8C21x45, CY8C22x45, CY8C24x23, CY8C24x94, CY8C28xxx

Software Version: PSoC® Designer™ 5.4

Related Application Notes: None

AN2034 shows how to use PSoC® 1 to read mechanical keypads. It covers matrix and common bus keypads, in both polled and interrupt modes.

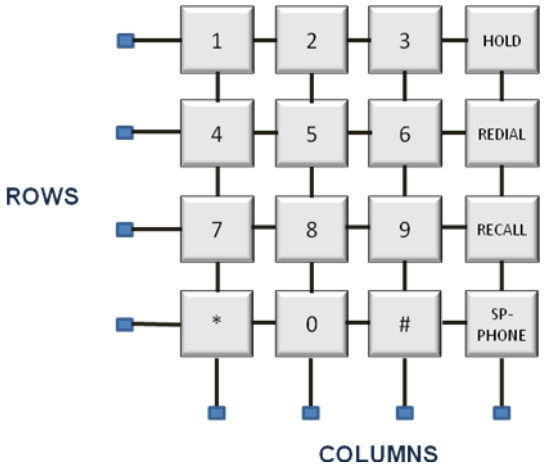
Contents

Introduction

Depending on how individual switches are connected, mechanical keypads are commonly available in two forms – matrix and common bus.

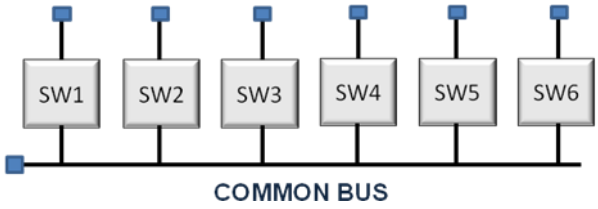
A typical matrix keypad – the telephone keypad - is shown in Figure 1. In this keypad, each switch connects a unique pair of row and column wires. The number of I/Os (pins) required equals ROWS + COLUMNS. The number of switches supported is ROWS x COLUMNS.

Figure 1. Matrix Keypad



In a common bus keypad, one terminal of each switch is connected to a common bus wire as shown in Figure 2.

Figure 2. Common Bus Keypad



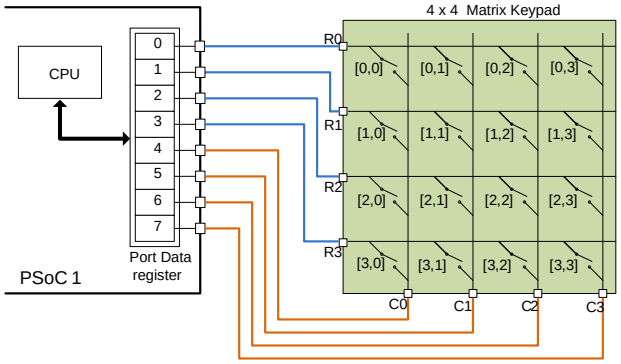
The number of I/Os required for a common bus keypad appears to equal the number of switches plus 1, however, a method exists that uses only one pin to read the common bus keypads.

This application note shows how to use PSoC 1 to read both the matrix and common bus keypads. The associated projects are created using the PSoC Designer software tool. If you are new to the PSoC 1 device or the PSoC Designer development tool, it is recommended that you review the online tutorials.

Matrix Keypad

This section focuses on a typical 4x4 matrix keypad, but you can use the same technique with a matrix keypad of any dimensions. A matrix keypad can be connected to the PSoC 1 without using any external components as shown in Figure 3.

Figure 3. Matrix Keypad Interface to PSoC 1



PSoC 1 pins are highly flexible and can be configured in many different modes, such as strong drive, resistive pull-up, resistive pull-down, open drain drive high/low, and high impedance. To avoid a floating input when no key is pressed, use the resistive pull-down mode for all row and column pins.

Algorithm

The standard algorithm for reading a matrix keypad is to drive each row high, one at a time, and sample the column lines. This technique enables detection of multiple key presses. However, scanning the entire keypad takes significant CPU time. If only a single key press needs to be detected, you can use the following algorithm instead:

- Drive all rows high simultaneously and read all columns
- Drive all columns high simultaneously and read all rows
- Combine the above data to determine which switch is closed

To understand how the algorithm works, consider the following example, where switch [1, 2] (row 1, column 2) is pressed. The algorithm reads the keypad in six steps.

1. Output b00001111 to the port. This drives all the rows high, leaving the columns passively pulled down.
2. Read the port. The driven pins 0 through 3 remain high, and because the switch [1, 2] is closed pin 6 is now high. The value read is b01001111.
3. Output b11110000 to the port. This drives all the columns high, leaving the rows passively pulled down.
4. Read the port. The driven pins 4 through 7 remain high, and because the switch [1, 2] is closed pin 1 is now high. The value read is b11110010.
5. Do a logical AND of the results of steps 2 and 4 to get the answer b01000010.
6. The upper 4 bits can be decoded as column 2 and the lower 4 bits as row 1.

C Function

The above algorithm can be implemented as a function keypad_scan(). It reads a 4x4 keypad connected to port 0 and returns the ANDed result, as shown in Code 1:

Code 1. Keypad Scan Function

```
/*
This is the port mapping for the keypad
ports.

p0.0 ---1----2----3----A
        |    |    |    |
p0.1 ---4----5----6----B
        |    |    |    |
p0.2 ---7----8----9----C
        |    |    |    |
p0.3 ---*----0----#----D
        |    |    |    |
      p0.4 p0.5 p0.6 p0.7
*/
unsigned char keypad_scan(void)
{
    unsigned char rows;
    unsigned char cols;

    /* Drive rows, read columns */
    PRT0DR = 0x0F;
    cols = PRT0DR;

    /* Drive columns, read rows*/
    PRT0DR = 0xF0;
    rows = PRT0DR;
```

```
    /* combine results */
    return (rows & cols);
}
```

Decoding a Key Press

The output of the keypad_scan() function is a single byte that shows the row or column status of the keypad. It is decoded as follows:

- No bits are set if no key is pressed
- A single bit in the upper nibble and a single bit in the lower nibble are set for a single key press
- Any other condition is a multiple key closure and is defined as not valid

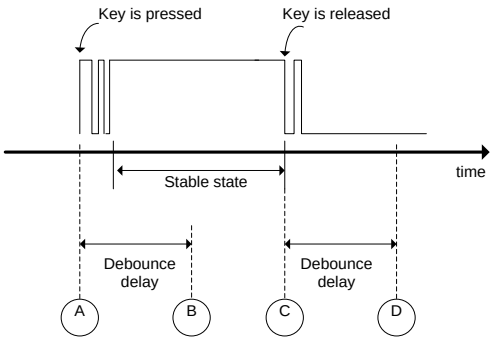
You will usually need to translate the row or column bits into a more useful form, for example an ASCII character corresponding to the key press. The most speed efficient way to do this is to use a lookup table (LUT) array. For a size efficient implementation, use switch case statements.

In this project, the LUT method is used: the byte output of the keypad_scan() function is used as an index to a 256-byte array. In the 4x4 keypad case, 16 elements in the array contain codes corresponding to valid key presses, and the remaining elements contain codes for "no key press" or "undefined". See Appendix A: 'C' Code for Matrix Keypad Scanning Using Interrupt Method for details.

Debounce Mechanism

When any key is pressed or released, due to mechanical structure, the contacts may make or break connection multiple times. Due to this, an oscillating waveform can be obtained at the row or column line as shown in Figure 4:

Figure 4. Key Bounce



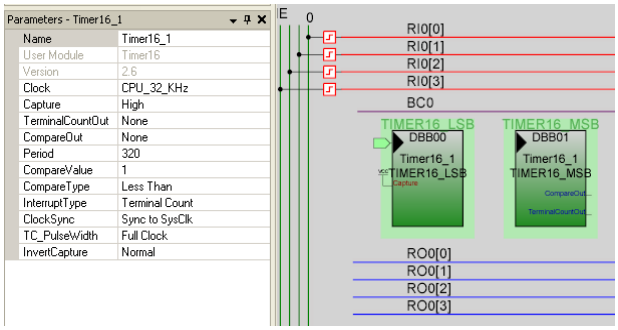
If the CPU reads the ports at a high rate, the bounce oscillations may be wrongly detected as multiple key presses. One way to handle this problem is to implement a delay from when the first edge is detected (A or C) to when the ports are read (B or D). This delay ensures that the signal at the port pins is stable during the CPU read operation. A delay of 10 ms is usually sufficient for most types of switches.

A delay can be implemented using a firmware or a hardware timer. The disadvantage of a firmware delay is that it requires the CPU. In an application where the CPU must do many tasks, it is better to generate the delay using a hardware timer. PSoC 1's configurable digital blocks make it possible to create such a timer; it is available in the form of a PSoC Designer user module named Timer.

To add a debounce delay using a hardware timer, do the following:

1. Place a Timer user module in the design, as shown in Figure 5:

Figure 5. Timer User Module



2. Configure the timer clock input and period to generate the required delay. As an example, a 16-bit timer is configured with a 32 kHz internal low-speed oscillator (ILO) input clock, with a period of 320 counts, to generate a delay of 10 ms.

   You can also use an 8-bit timer, which consumes one less digital block, with a lower input clock frequency to generate the same delay. The input clock to a timer can be derived using internal clock dividers VC1, VC2, and VC3.

3. The Timer user module can generate an interrupt, and you can call the keypad scanning function from the Timer interrupt service routine (ISR). This will cause the CPU to read the ports after a debounce delay.

Making the Scanning Process Polled or Interrupt Driven

The scanning algorithm in the keypad_scan() function can be executed once the key press is detected. There are two ways to call the function and detect key activity - polling and interrupt.

With the polling method, the CPU periodically scans the keypad to check if any key is pressed. The CPU can either read the keypad in a while loop or can be interrupted at regular intervals using a timer. The scan period must be greater than the debounce delay and less than the minimum key press time. The problem with this method is that most of the time the keys are not pressed, and doing a scan during those times is a waste of CPU cycles. A better way is to do a scan only when a key press event is detected.

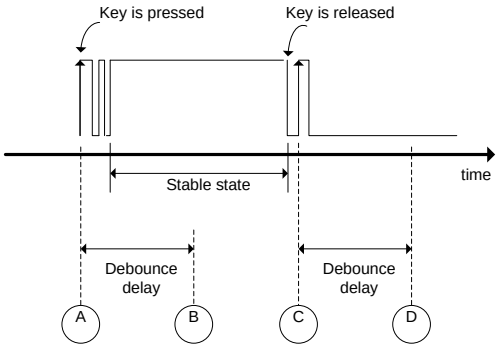
With the interrupt method, the CPU can be interrupted on a key press. The ISR then starts the timer for a debounce delay and a keypad scan is done at the timer interrupt.

To configure PSoC for an interrupt-based method, do the following:

- Configure the four lower (row) pins (see Figure 3) as inputs, with resistive pull-down
- Drive the four upper (column) pins high
- Write a GPIO interrupt handler that enables the timer, for a debounce delay
- Enable the GPIO interrupt

The row pins stay low when no key is pressed. On a key press, the rising edge signal is produced on one of the row lines which trigger an interrupt. The GPIO ISR enables the timer for a debounce delay. After the delay, the Timer ISR does the scan. See Figure 6 for more details.

Figure 6. Interrupt and Debounce Action



A — Key is pressed. Rising edge signal interrupts CPU. CPU starts the timer for debounce delay

B — Debounce delay elapsed. Timer triggers interrupt to the CPU to read the ports to identify the key pressed.

C — Key is released. Rising edge signal interrupts CPU once again. CPU starts the timer for debounce delay

D — Debounce delay elapsed. Timer triggers interrupt to the CPU to read port once again. As key is released, no key press will be detected.

* Note that switch may not bounce during switch release and C and D events may not occur.

The interrupt method also helps to reduce the power consumption by allowing the device to be put in sleep mode. When any key is pressed, the resulting interrupt can wake the device.

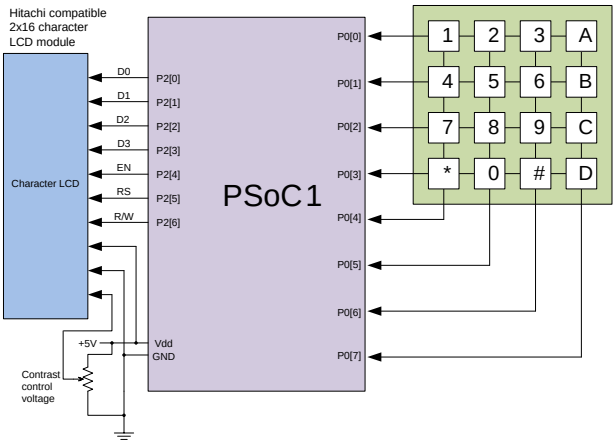
Note that the PSoC 1 interrupt system combines the interrupt signals from all of the pins in the device to generate a single interrupt. Thus, any key press will trigger a common interrupt. However, any other external event will also trigger the same interrupt. To handle this you may need to have the ISR read other ports to determine the cause of the interrupt.

Example Project

The PSoC Designer project “MatrixKeypad”, based on PSoC 1 CY8C27443, is provided. The project uses the interrupt method and is configured for the external connections shown in

Figure 7. In the project, the closed key is displayed on a Hitachi-compatible 2x16 character LCD. The number of times any key is pressed is also displayed to evaluate the debounce logic. If the debounce logic is working correctly, you will notice that the count increments only once on each key press. See Appendix A: ‘C’ Code for Matrix Keypad Scanning Using Interrupt Method for the application firmware.

Figure 7. External Connections for Testing "MatrixKeypad" Project



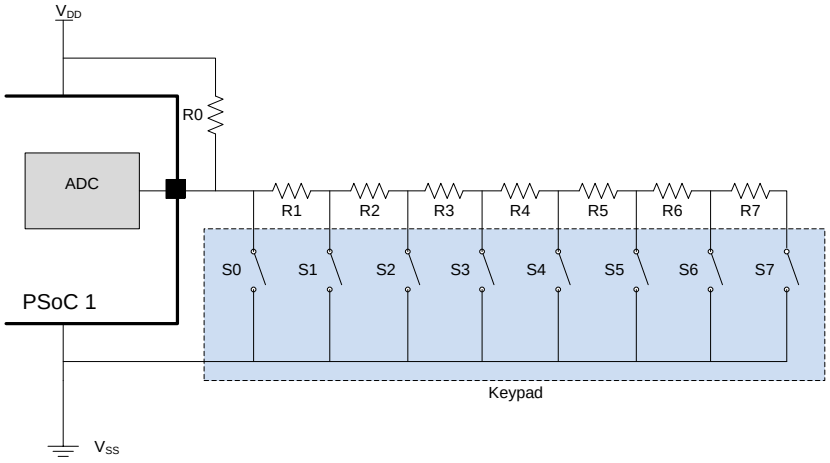
Common Bus Keypad

This section shows how to read an 8-key common bus keypad using only one PSoC pin. The logic behind the single pin solution is to generate a distinct analog voltage on a key press and to decode it using the ADC. The block diagram of the solution is given in Figure 8.

A switch-resistor network forms a voltage divider connected between the power supply (V_{DD}) and the ground (V_{SS}). The PSoC 1 ADC is configured to read the input voltage from V_{DD} to V_{SS}. The resistor values are chosen so that when a particular switch is pressed, the input voltage to the ADC is within a certain range. A simple algorithm can be created that compares the ADC result with the predetermined codes to identify the key pressed.

When multiple keys are pressed, the one which is closer to the pin takes priority; the resistor network beyond the closed key is bypassed. In Figure 8, S0 has the highest priority and S7 has the lowest priority.

Figure 8. Block Diagram - Common Bus Keypad Interface to PSoC



PSoC 1 Implementation

PSoC Designer provides different types of ADCs that are implemented using generic analog blocks in PSoC 1.

The resolution of the ADC is an important parameter for this application as it determines the number of switches that can be interfaced. Ideally, with an N-bit ADC 2^N switches can be read. But due to the tolerances of the resistors, it is significantly lower.

To read an 8-switch keypad, a 6-bit SAR ADC is sufficient. The SAR ADC is available in the form of a user module in PSoC Designer. This user module has the following features:

- Consumes one PSoC 1 switched capacitor (SC) analog block
- Produces a 1 byte 2's complement output, ranging from -32 to +31

Table 1. SAR ADC Input to Output Mapping

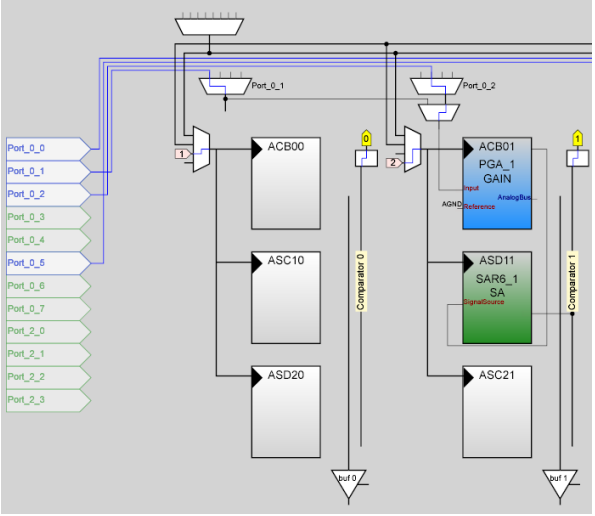
Input Voltage (V_{IN})	Digital Output	Comments
AGND (analog ground)	00h	RefMux setting in Global resources tab of PSoC Designer decides V_H and V_L and AGND.
AGND < V_{IN} < V_H	00h to 1Fh	
V_L < V_{IN} < AGND	E0 to FFh	

The input impedance of the SAR ADC is in the kΩ range; it depends on the frequency of the clock to the SC block. To avoid errors due to low input impedance, place a programmable gain amplifier (PGA) module between the SAR ADC and the external network. The PGA uses a PSoC 1 continuous time (CT) analog block which has high input impedance.

Using PSoC Designer, configure the PSoC 1 as follows:

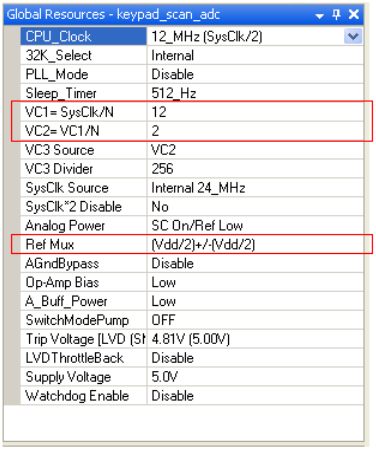
- Place the PGA and SAR ADC user modules on the design
- Route the PGA input to a pin connected to the external switch-resistor network, as shown in Figure 9.

Figure 9. User Modules Placement and Routing



- Route the PGA output to the SAR ADC input
- Set the PGA gain to 1 in the PGA module properties
- Set the analog column clock to VC2, with the frequency between 128 kHz and 1.33 MHz in the global resource settings. This clock controls the switching frequency of the SC block.
- In the global resources tab, set the VC2 frequency to 1 MHz by setting the VC1 divider to 12 and the VC2 divider to 2, as shown in Figure 10. Note that the system clock (SysClk) of the PSoC 1 device is 24 MHz.
- Also in the global resources tab, set the RefMux option to (V_{DD}/2) +/- (V_{DD}/2). This sets the SAR ADC measurement range as 0 to V_{DD}, and the AGND voltage to V_{DD}/2.

Figure 10. Global Resource Settings



SAR ADC Input Voltages

When mapped uniformly for a V_{DD} of 5.0 V, each switch produces a nominal increase of 5.0 V / 8 = 0.625 V, or a 12.5 percent change in the ADC input voltage. Table 2 provides the input voltage range for the ADC for each switch. The midpoint of each range is the theoretical ADC input voltage neglecting all error sources such as resistor tolerances, ADC error, noise, and so on.

Table 2. Mapping Switch Press to the Voltage Generated

Switch	Voltage Range at Input	Byte Range	Resistor Value
S0	0 V – 0.3125 V	[E0 – E4]	-
S1	0.3125 V – 0.9375 V	[E4 – EC]	R1 = 1.5 kΩ
S2	0.9375 V – 1.5625 V	[EC – F4]	R2 = 1.8 kΩ
S3	1.5625 V – 2.1875 V	[F4 – FC]	R3 = 2.7 kΩ
S4	2.1875 V – 2.8125 V	[FC – 04]	R4 = 4.3 kΩ
S5	2.8125 V – 3.4375 V	[04 – 0C]	R5 = 6.8 kΩ
S6	3.4375 V – 4.0625 V	[0C – 14]	R6 = 13 kΩ
S7	4.0625 V – 4.6875 V	[14 – 1A]	R7 = 33 kΩ
No Switch	5.0 V	1F	-

If none of the switches are closed, the input to the ADC is equal to V_{DD}. When S0 is closed, current flows from V_{DD} to V_{SS} through R0 and S0. A value of 10 kΩ was chosen for R0 to limit the maximum supply current to 0.5 mA. The remaining resistors are calculated based on V_{DD} = 5 V, R0 = 10 kΩ, and a nominal step size of 0.625 V for each switch. The following section explains how to select the resistors.

Determining Resistor Values

Basic resistor voltage divider analysis can be used to determine the resistor values. Consider the case when switch S1 is pressed (see Figure 8)

$$V_{DD} \times \frac{R_1}{R_1 + R_0} = 0.625\ V$$

Solving for R1 and inserting nominal values for V_{DD} and R0 gives

$$R_1 = \left(\frac{0.625\ V \times 10\ k\Omega}{5.0\ V - 0.625\ V}\right) = 1.43\ k\Omega$$

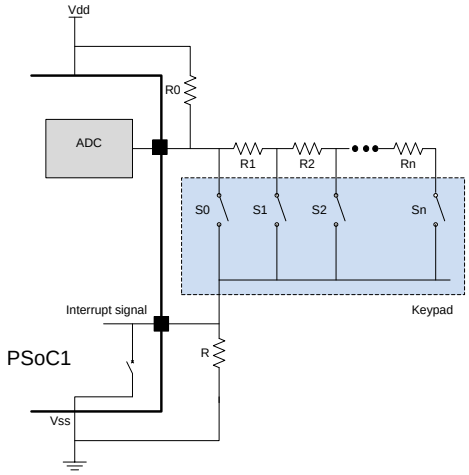
The closest ±5% standard value resistor is 1.5 kΩ. With R1 = 1.5 kΩ and R0 = 10 kΩ, the typical ADC input voltage with S1 closed is 0.652 V. This error is acceptable since it falls within the allowed voltage range for SW1, that is, 0.3125 V to 0.9375 V. Moreover, with the worst case resistor tolerances (R0 = 10 kΩ – 5% and R1 = 1.5 kΩ + 5%, or R0 = 10 kΩ + 5% and R1 = 1.5 kΩ – 5%), the input voltage deviates by just 0.059 V from the ideal value.

The same type of analysis can be used to determine the remaining resistor values, which are provided in Table 2.

Interrupt Driven Keypad Reading

As seen in the section Making the Scanning Process Polled or Interrupt Driven, it is better to read the ADC on an interrupt when any key is pressed rather than reading the ADC continuously. This saves CPU bandwidth as well as device power consumption if sleep mode is used. To make the process interrupt driven, one more pin must be used for generating an interrupt signal. See Figure 11 for external connections.

Figure 11. Interrupt Driven Scanning Process



The interrupt pin is pulled low using a resistor (R). When any key is pressed, the rising edge signal is developed across R causing an interrupt to the device. The value of the resistor R must be high enough so that the voltage developed across it, on any key press, is greater than the V_{IH} specification for a GPIO (minimum 2.1 V).

After an interrupt is generated, the interrupt pin is grounded by writing a '0' to the port data register. This causes the resistor R to be bypassed while reading the voltage using the ADC. Note that the lower end of the switches is not exactly 0 V - it is lifted above the ground by a few millivolts depending on the sink current value. Using the higher value of resistor R0 limits the sink current and thus the voltage lift. For this application, use R0 of 10 kΩ or higher.

Debouncing

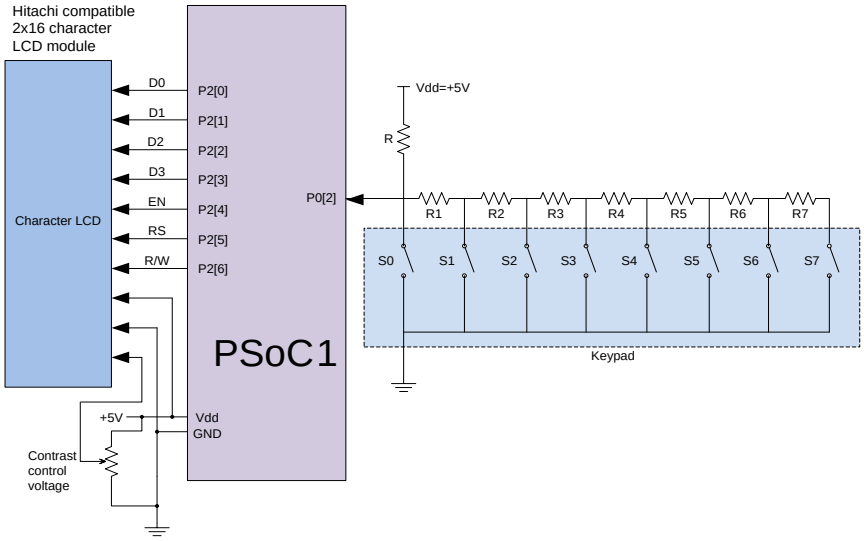
It is possible for a switch press or release to occur during an ADC conversion, which would result in an erroneous output. To avoid these errors, you can implement a debounce mechanism similar to that for the matrix keypad:

- Configure the timer user module to generate a delay of at least 10 ms
- If an interrupt method is used, start the timer when the GPIO interrupt is detected. For the polling method, start the timer when the ADC reads a value different from the "no keys pressed" value.
- At the timer interrupt, read the ADC. If the ADC reading is same as the previous reading, a key can be assumed to be pressed and stable.

Example Project

The project "CommonBusKeypad", based on the CY8C27443 PSoC 1 device, uses the polling method. The complete code is also provided in Appendix B: 'C' Code for Common Bus Keypad Using Polling Method. The project can be tested with the external connections shown in Figure 12. In the project, key presses are displayed on a Hitachi-compatible 2x16 character LCD.

Figure 12. External Connections for Testing “CommonBusKeypad” Project



Use the resistor values R and R1 to R7 as shown in Table 2.

Summary

This application note shows easy methods to read matrix type and a common bus type keypads using PSoC 1. It also explains the polling and the interrupt mechanisms for keypad scanning as well as debounce logic.

About the Author

Name:	Rajiv Badiger
Title:	Applications Engineer
Background:	BE Electronics and Telecommunication
Contact:	rjvb@cypress.com

Appendix A: 'C' Code for Matrix Keypad Scanning Using Interrupt Method

```
#include <m8c.h>  // part specific constants and macros
#include "PSoCAPI.h"  // PSoC API definitions for all User Modules


/*****************Variables*************************/
/* LUT for storing keys. The size of the LUT should be equal to 2 ^ (NUMBER OF ROWS +
COLUMNS). Row and column information from the scanning function is used as index to this LUT
for finding the key. Value 0x20 indicates invalid key. */

static const keypad_LUT[256] =
{/*   0     1     2     3     4     5     6     7
      8     9     A     B     C     D     E     F  */
/*0*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*1*/0x20,  '1',  '4', 0x20,  '7', 0x20, 0x20, 0x20,
     '*', 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*2*/0x20,  '2',  '5', 0x20,  '8', 0x20, 0x20, 0x20,
     '0', 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*3*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*4*/0x20,  '3',  '6', 0x20,  '9', 0x20, 0x20, 0x20,
     '#', 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*5*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*6*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*7*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*8*/0x20,  'A',  'B', 0x20,  'C', 0x20, 0x20, 0x20,
     'D', 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*9*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*A*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*B*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*C*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*D*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*E*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
/*F*/0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20
};

/* These two variables are used to store the row number and the column number */
BYTE rows,cols;

/* This variable holds the count for number of key press */
BYTE InterruptCounter = 0;

/* This variable keeps the scanning process disabled till previously decoded key is read */
unsigned char processing = 0;
```

```
/* These variable stores the key number */
unsigned char key = 0x20;
/*****************************************************/


/***************Functions*****************************/

/* Handles keypad function- Checks the key pressed and debounces key */
void KeyPadHandler(void);

/* Keypad scanning routine */
unsigned char keypad_scan(void);

/* Initializes user modules used in the design */
void InitModules(void);

/* Interrupt handler for GPIO interrupt */
void GPIOInterrupt(void);
/*****************************************************/


void main(void)
{
	/* Initialize variables and user modules */
	InitModules();

	/* Enable Global Interrupt */
	M8C_EnableGInt;

	while(1)
	{
		/* Check the Keypad status and update LCD on valid key press */

		if(key != 0x20)
		{
			/* Valid key- display the result on LCD */

			/* Set LCD cursor position */
			LCD_1_Position(0, 7);

			/* Display pressed keypad button */
			LCD_1_WriteData(key);

			/* Set LCD cursor position */
			LCD_1_Position(1, 12);

			/* Display nuymber of times key is pressed so far */
			LCD_1_PrHexByte(InterruptCounter);

			/* Reset key to default value */
			key = 0x20;

			/* Enable keypad scanning process */
			processing = 0;
		}

		/* Do other tasks */
	}
}
```

```
/* This function initializes all the user modules used in the design */
void InitModules(void)
{
	/* Initialize LCD */
	LCD_1_Start();
	LCD_1_Position(0,0);
	LCD_1_PrCString("Switch:");
	LCD_1_Position(1,0);
	LCD_1_PrCString("Interrupts:");

	/* Initilize timer- 10ms delay */
	Timer16_1_WritePeriod(320);
	Timer16_1_EnableInt();

	/* Enable GPIO interrupt */
	INT_MSK0 |= 0x20;

	/* Write 1's at column lines */
	PRT0DR |= 0xF0;
}

/*
This is the port mapping for the keypad ports.

	  p0.4 p0.5 p0.6 p0.7
	  |    |    |    |
p0.0 ---1----2----3----A
	  |    |    |    |
p0.1 ---4----5----6----B
	  |    |    |    |
p0.2 ---7----8----9----C
	  |    |    |    |
p0.3 ---*----0----#----D

*/

/* This function scan the keypad and identifies the key pressed */
unsigned char keypad_scan(void)
{
	BYTE key_result;

	/* Drive rows */
	PRT0DR = 0x0F;

	/* Read columns */
	cols = PRT0DR;

	/* Drive columns */
	PRT0DR = 0xF0;

	/* Read rows */
	rows = PRT0DR;

	/* Combine results */
	key_result = rows & cols;

	/* Get the key number from LUT */
	return(keypad_LUT[key_result]);
}
```

```
/* This is the handler for GPIO interrupt. Debounce timer is started on GPIO interrupt */
#pragma interrupt_handler GPIOInterrupt
void GPIOInterrupt(void)
{
	/* Check if previous key is read. If yes, start the timer for debounce delay */
	if(processing == 0)
	{
		/* Avoid enabling timer next time when previous key is not read */
		processing = 1;

		/* Disable GPIO interrupt */
		INT_MSK0 &=~ 0x20;

		/* Start Timer */
		Timer16_1_Start();
	}
}


/* This is the handler for the timer interrupt. This ISR calls the scanning algorithm */
void TimerInterrupt(void)
{
	/* Stop the timer */
	Timer16_1_Stop();

	/* Get the key */
	key = keypad_scan();

	/* Check for valid key press */
	if(key != 0x20)
	{
		/*************************************************************/
		/* Add code here to take immediate action within the ISR */


		/*************************************************************/

		/* Increment interrupt counter */
		InterruptCounter ++;
	}
	else
	{
		/* Enable starting the timer on GPIO interrupt */
		processing = 0;
	}

	/* Clear GPIO posted interrupt */
	INT_CLR0 &=~ 0x20;

	/* Enable GPIO interrupt */
	INT_MSK0 |= 0x20;


}
```

Add LJMP instruction in PSoC_GPIO_ISR function in *PSoCGPIOINT.asm* file as shown in the following code.

```
PSoC_GPIO_ISR:

    ;@PSoC_UserCode_BODY@ (Do not change this line.)
    ;---------------------------------------------------
    ; Insert your custom code below this banner
    ;---------------------------------------------------
    ljmp _GPIOInterrupt
    ;---------------------------------------------------
    ; Insert your custom code above this banner
    ;---------------------------------------------------
    ;@PSoC_UserCode_END@ (Do not change this line.)

    reti
```

Add LCALL instruction in Timer16_1_ISR function in *Timer16_1NT.asm* file as shown below.

```
_Timer16_1_ISR:

    ;@PSoC_UserCode_BODY@ (Do not change this line.)
    ;---------------------------------------------------
    ; Insert your custom assembly code below this banner
    ;---------------------------------------------------
    ;   NOTE: interrupt service routines must preserve
    ;   the values of the A and X CPU registers.

    ;---------------------------------------------------
    ; Insert your custom assembly code above this banner
    ;---------------------------------------------------

    ;---------------------------------------------------
    ; Insert a lcall to a C function below this banner
    ; and un-comment the lines between these banners
    ;---------------------------------------------------

    PRESERVE_CPU_CONTEXT
    lcall _TimerInterrupt
    RESTORE_CPU_CONTEXT

    ;---------------------------------------------------
    ; Insert a lcall to a C function above this banner
    ; and un-comment the lines between these banners
    ;---------------------------------------------------
    ;@PSoC_UserCode_END@ (Do not change this line.)

    reti
```

Appendix B: 'C' Code for Common Bus Keypad Using Polling Method

```
#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all User Modules

/* Timer State */
#define ON 0x1
#define OFF 0x0

/* Invalid Key identifier */
#define INVALID_KEY 0x20

/* Functions */
char Get_ADC_Data(void);
void GetKeyNumber(void);

/* Variables */
BYTE Key = INVALID_KEY, TimerState = OFF;
char ADC_Data,New_ADC_Data;


void main(void)
{
	/* Enable Timer Interrupt */
	Timer16_1_EnableInt();

	/* Enable Global Interrupt */
	M8C_EnableGInt ;

	/* Start PGA operation in high power mode */
	PGA_1_Start(PGA_1_HIGHPOWER);

	/* Start ADC operation */
	SAR6_1_Start(SAR6_1_HIGHPOWER);

	/* Start LCD and Initialize Display */
	LCD_1_Start();
	LCD_1_Position(0,0);
	LCD_1_PrCString("Switch:");

	while(1)
	{
		/* If timer is OFF (debounce is not in process), check the "key" */
		if(TimerState == OFF)
		{
			/* If "Key" contains valid value, then display on LCD */
			if(Key != INVALID_KEY)
			{
				LCD_1_Position(0,7);
				LCD_1_PrHexByte(Key);

				/* Invalidate Key */
				Key = INVALID_KEY;
			}
			else /* No Key Pressed */
			{
				LCD_1_Position(0,7);
				LCD_1_PrCString("--");
			}

			/* Get ADC data */
```

```
                    ADC_Data = SAR6_1_cGetSample();

                    /* Check if any key is pressed */
                    if(ADC_Data < 0x1C || ADC_Data > 0xE0)
                    {
                        /* Start Timer */
                        Timer16_1_Start();
                        TimerState = ON;
                    }
                }

                /* Do other tasks */
        }
}

/* Interrupt Handler for Timer Interrupt */
void TimerInt(void)
{
        BYTE Delta = 0;

        /* Take a second sample, compare with the first to test
        data validity in the presence of possible switch bounce */
        New_ADC_Data = SAR6_1_cGetSample();

        /* Compare the ADC values and get the difference */
        if(ADC_Data > New_ADC_Data)
        {
                Delta = ADC_Data - New_ADC_Data;
        }
        else
                Delta = New_ADC_Data - ADC_Data;

        /* If difference in readings is less than 2, then decode the key */
        if(Delta < 2)
        {
                /* If closely matched, decode the key */
                GetKeyNumber();
        }

        /* Stop the timer */
        Timer16_1_Stop();
        TimerState = OFF;

}

/* This function decodes the key.  See Table 2 of application note for ADC result to Key
mapping information */
void GetKeyNumber(void)
{
        if((New_ADC_Data > 0xE0) && (New_ADC_Data <= 0xE4))
                Key = 0;
        else if((New_ADC_Data > 0xE4) && (New_ADC_Data <= 0xEC))
                Key = 1;
        else if((New_ADC_Data > 0xEC) && (New_ADC_Data <= 0xF4))
                Key = 2;
        else if((New_ADC_Data > 0xF4) && (New_ADC_Data <= 0xFC))
                Key = 3;
        else if((New_ADC_Data > 0xFC) && (New_ADC_Data <= 0xFF))
                Key = 4;
        else if((New_ADC_Data >= 0x0) && (New_ADC_Data <= 0x04))
                Key = 4;
```

```
        else if((New_ADC_Data > 0x04) && (New_ADC_Data <= 0x0C))
                Key = 5;
        else if((New_ADC_Data > 0x0C) && (New_ADC_Data <= 0x14))
                Key = 6;
        else if((New_ADC_Data > 0x14) && (New_ADC_Data <= 0x1A))
                Key = 7;
        else
                Key = INVALID_KEY;
}
```

Add LCALL instruction in Timer16_1_ISR function in *Timer16_1NT.asm* file as shown in the following code.

```
_Timer16_1_ISR:

    ;@PSoC_UserCode_BODY@ (Do not change this line.)
    ;---------------------------------------------------
    ; Insert your custom assembly code below this banner
    ;---------------------------------------------------
    ;   NOTE: interrupt service routines must preserve
    ;   the values of the A and X CPU registers.

    ;---------------------------------------------------
    ; Insert your custom assembly code above this banner
    ;---------------------------------------------------

    ;---------------------------------------------------
    ; Insert a lcall to a C function below this banner
    ; and un-comment the lines between these banners
    ;---------------------------------------------------

    PRESERVE_CPU_CONTEXT
    lcall _TimerInt
    RESTORE_CPU_CONTEXT

    ;---------------------------------------------------
    ; Insert a lcall to a C function above this banner
    ; and un-comment the lines between these banners
    ;---------------------------------------------------
    ;@PSoC_UserCode_END@ (Do not change this line.)

    reti
```

Document History

Document Title: AN2034 – PSoC® 1 - Reading Matrix and Common Bus Keypads

Document Number: 001-40409

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1532004	OGNE	10/02/2007	New publication of existing application note.
*A	2640952	JVY	01/20/2009	Updated content. Added part numbers CY8C20x34, CY8C21x23, CY8C21x34, CY8C23x33, CY8C24x23A, CY8C24x94, CY8C27x43, and CY8C29x66.
*B	3312765	OWEN	07/19/2011	Added section about LED display, converted code to C from assembly, updated associated project similarly.
*C	3416292	RJVB	10/24/2011	Added common bus keypad interface details, introduced debounce logic and interrupt mechanism. Updated template
*D	3433993	RJVB	11/09/2011	Modified title and updated project code in Appendix A and B
*E	4463001	ARVI	08/01/2014	Updated example projects to PSoC Designer v5.4
*F	5708837	AESATMP9	04/26/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709