Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.



THIS SPEC IS OBSOLETE

Spec No: 001-17215

Spec Title: PSOC(R) 1 POWER MANAGEMENT - VOLTAGE MONITORING AND SEQUENCING WITH PSOC AND I2C - AN2379

Sunset Owner: Praveen Kumar Murugesan (prku)

Replaced By: 001-78646

PSoC® 1 Power Management – Voltage Monitoring and Sequencing with PSoC and I2C

AN2379

Author: Ernie Buterbaugh
Associated Project: Yes
Associated Part Family: CY8C24x23A, CY8C27x43, CY8C29x66
Software Version: PSoC® Designer™ 5.1
Associated Application Notes: AN2154

OBSOLETE

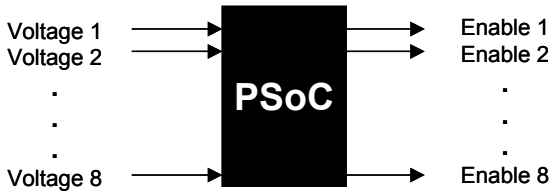
Application Note Abstract

Many systems require a variety of voltages to power the ASICs, FPGAs, and standard logic found in today's designs. To further complicate the design, some of these devices require voltages to power-on in a specific sequence, lest damage to the chip will occur. There are a variety of voltage monitoring and sequencing devices available today. However, they can be expensive and often do not meet the needs of the design since they offer little or no programmability. A PSoC device is an excellent choice for monitoring and sequencing the voltages. With its analog structure and programmability, the PSoC can be a perfect fit for the most stringent designs. AN2379 is an updated and enhanced version of AN2154 and includes, by request, I2C.

Introduction

This Application Note describes a system that uses eight different voltages. The PSoC device monitors each of the eight rails and controls the voltage-enables such that they power-on in sequence. Subsequent rails are activated only after the previous rail rises to a minimal voltage level. All the voltages are constantly monitored and if any voltage falls below a minimum value, the appropriate enables are turned off. The configuration can be easily adapted to satisfy other designs.

Figure 1. Voltage Monitor

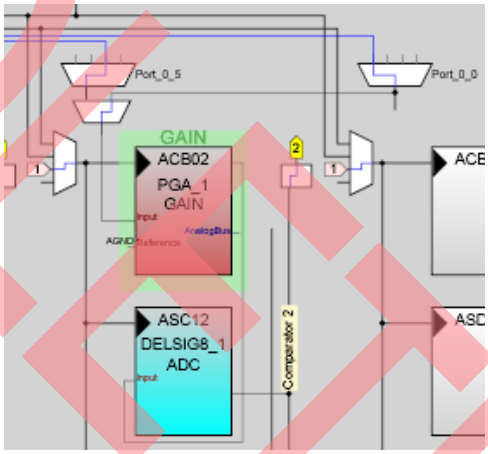


Monitoring Voltages

The first step is to monitor all the voltage rails. Burdened with large capacitors, voltage rails tend to change very slowly. This allows a variety of options for capturing the value.

The PSoC has resources for voltage comparators and also A/D converters. To allow the most flexibility, we will choose the ADC method for this implementation.

Figure 2. PGA and ADC Placement



There are several ADCs from which to choose. The simple SAR6 is an easy-to-use ADC and will suffice for many applications. This project will use the more advanced DELSIG8 8-bit delta sigma ADC for its increased resolution and accuracy. A larger 11- or 12-bit delta sigma ADC may be substituted if more accuracy is desired.

With the DELSIG8, as is the case with all the delta sigma A/Ds in PSoC, an interrupt occurs whenever a conversion is complete and a digital value is ready.

Since there are eight analog inputs that we need to monitor, we will take advantage of the 8:1 analog input mux and use only one ADC. This allows us to sequence through each rail and capture the value. The 8:1 mux feeds into a programmable gain amplifier (PGA) and then the ADC. The default position of the mux set in the Device Editor is Port0[0], which corresponds to Voltage_1.

The gain of the PGA is set to 1x. However, depending on the value of the voltage rail, the PGA can be used to increase the gain and hence, increase the resolution of the ADC. Because of the programmable nature of the PSoC, the gain can be selected on a rail-by-rail basis. The gain of the PGA can also be set in the same interrupt service routine (ISR) that changes the analog mux.

Note that the input voltages have to be scaled down to 2.6V for a RefMux of Bandgap ± Bandgap and to V_{cc} for a RefMux of $V_{cc}/2$ ± $V_{cc}/2$ by external resistor dividers for proper operation. The project demonstrates the operation with voltage steps from 0.75V to 2.25V in 0.25V steps. The thresholds can be modified according to system requirements.

Referring to the ISR code in Figure 3, every time the ISR is entered, i.e., the ADC has completed an acquisition, the Trash_Can variable is decremented and checked. If the value is higher than zero, the ADC has not been sampled enough times since the analog mux had switched and the ISR returns without setting the ADC data valid flag. The DELSIG ADC result at any time includes data from two previous samples. When the input to the Delta Sigma converter is multiplexed two samples must be processed through to get rid of old data before the current data is valid. A trash counter in this design is used to discard the first three samples until data is passed to the *main.c* routine. Also, if the data that was captured before has not been taken by the main routine, do not update with the new data yet – just return.

We are now at the point where the data is good and is available in Register A. The value is in 2's complement and this may be the desired form. However, if 1's complement better suits the needs of the design, a simple conversion method is to XOR the MSB bit as shown in the example.

The Trash_Can variable is reset next in the ISR code since we are about to change the analog mux position.

We also save the current mux position that corresponds to the captured data and then specify the next mux position. If less than eight inputs are used, the next mux position calculation is adjusted to switch to only the valid positions. The 8:1 mux is made from two 4:1 muxes; in this case, AMUX3 and AMUX2. We need to select the position in the AMUX2 or AMUX3 for the selected voltage and choose which of the two muxes is selected. While the code snipet in Figure 3 show a 'call AMUX4_n_InputSelect' to change the mux position, the actual ISR code in the code example does not use this function call but rather write the bits in the control register directly. This eliminates any memory paging conflicts with the Large Memory Model (LMM) PSoC. Writing to the ABF_CR0 register in the PSoC selects which mux is gated to the PGA. Note that this register is in register Bank 1 and we use the M8C_SetBank1 macro to access it. If the gain of the PGA is to change, the code (a call to PGA1_Set_Gain) is placed in this section as well.

Figure 3. ADC ISR

```
;;;   START ISR APPNOTE MODs
   push x
   dec [_Trash_Can]     ; throw away after
mux switch
   jnz end_ret
   cmp [_ADCFlag], 1    ; data not yet taken
   jnz getdata
   ; data was ready but main loop didn't
take it yet
   ; restore trash counter
for next pass
   inc [_Trash_Can]
   jmp end_ret

getdata:        ;; Data in ADC is good

   xor a,80H        ; Convert to 1's  comp
   mov [_ADCData], a      ; Save data
   mov
[_Trash_Can], 04H  ; Reset trash counter
   mov [_ADCFlag],1       ; Data available
flag
   mov a, [_MUXPosition]  ; Save mux
position
   mov [_MUXData], a
   inc a                  ; now set new mux
position
   and a, 07H       ; only 0-7 positions
   mov [_MUXPosition],A   ; save away

;; ---------------------------------------
;; This code is to select one of the 8
analog ;; input ports
;; A has value 0-7 selecting which port
;; Value: 0: Port 0_0  Volt_1     AMUX3
;; Value: 1: Port 0_2  Volt_2     AMUX3
;; Value: 2: Port 0_4  Volt_3     AMUX3
;; Value: 3: Port 0_6  Volt_4     AMUX3
;; Value: 4: Port 0_1  Volt_5     AMUX2
;; Value: 5: Port 0_3  Volt_6     AMUX2
;; Value: 6: Port 0_5  Volt_7     AMUX2
;; Value: 7: Port 0_7  Volt_8     AMUX2

;; ---------------------------------------

   cmp A,04H          ; If mux <4 then do
amux3
   jnc amux2

amux3:
   'call AMUX4_3_InputSelect' ; set mux in
column 3
M8C_SetBank1
   or reg[ABF_CR0], 0b01000000  ;  Column 3
   M8C_SetBank0
   jmp end_ret
```

OBSOLETE

```
amux2:

    'call AMUX4_2_InputSelect'
; set mux in column 2

    M8C_SetBank1
    and reg[ABF_CR0], 0b10111111 ;
Column 2
    M8C_SetBank0

end_ret:
pop x
;;
;; END ISR APPNOTE MODs
```

Sequencing Voltages

Now that the voltages are known, they can be checked for validity and the appropriate enables turned on or off.

The *main.c* section of the code, as the extension suggests, is written in C. C allows the code to be easily maintained and modified for a variety of system needs. The approach used in this design takes advantage of the switch-case statement to build a state machine. The state machine concept provides the ability to add states as well as allows voltage enables to be sequenced in a specific order and easily modified.

Many power systems require a delay between the times a rail powers on and the subsequent power enable. The state machine design allows for this delay to be added with ease. In this example, a 10-ms delay is used between the detection of the rail and the activation of the next rail. A function called Start_Timer() allows the program to set the delay in 1-ms increments, simply by passing the amount of delay time in ms to the subroutine. The timer is implemented in a digital block and causes an interrupt once the time passed to the routine expires.

As each power rail turns on and reaches the minimum value, the state machine will delay 10 ms and then turn on the next rail. This continues until all rails are on and the state machine reaches the idle state. Should any voltage fall below the minimum value, all the enables that were turned on after that power rail had come up, are turned off. The state machine then continues at that position in the sequence, waiting for the rails to become active. A small hysteresis of two counts is introduced during testing if the voltage has fallen below threshold. This eliminates any unwanted oscillations in the outputs.

The code has provisions for a minimum and a maximum voltage value. This provides the ability to window the valid range – that is, monitor for under voltage as well as over voltage. The demonstration code only uses the minimum value. Adding statements to flag and shut down when the maximum is exceeded can easily be done.

Resolution

So how accurately can the PSoC capture values? Of course that depends on a variety of factors such as the size of the ADC, the reference voltage, the gain of the PGA, and a few other factors. Let us take a couple of examples and show how the numbers compare.

There are several choices for the RefMux setting but let us take a look at two: BG +/- BG and Vcc/2 +/- Vcc/2. The bandgap (BG) reference voltage of the PSoC is 1.3V with a tolerance of 1%.

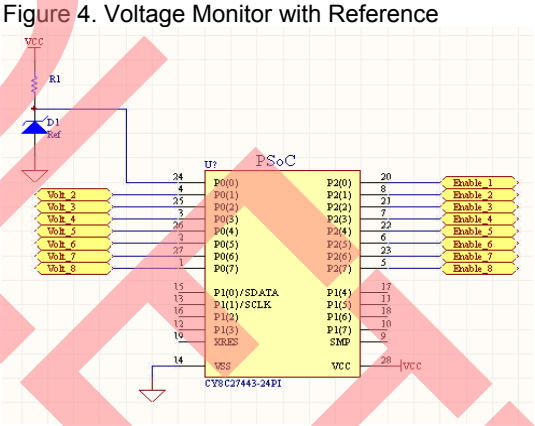
The BG +/- BG setting translates into a range of 0 to 2.6V. If you use the 8-bit ADC, this yields 2.6V / 255 = 10.196 mV per tick. If the Vcc/2 +/- Vcc/2 is used and Vcc is 3.3V, then the range to the ADC is 0 to 3.3V. This yields 3.3V / 255 = 12.941 mV per tick. In the Vcc case, the accuracy of the reference is subject to the variation in the Vcc supply. This is often greater than the 1% of the BG reference voltage.

Offset Voltages

Like all amplifiers and ADCs, there is a DC offset voltage that is added to the incoming signal. These offsets are not fixed but can vary and thus act as voltage error. For 3.3V operation, the PGA has a maximum offset voltage of 3.5 mV and the ADC is 5 mV. For many voltage-monitoring systems, this amount of error is inconsequential and can be ignored.

However, for those systems where the DC offset needs to be minimized, the PSoC can be *calibrated* to zero-out the DC offset. This is accomplished by bringing in a reference voltage through the same path as the voltages we are monitoring.

Figure 4. Voltage Monitor with Reference



The value captured from the ADC when sampling the reference voltage is now known to be exactly the reference voltage – i.e., the ADC is *calibrated*. If using the BG +/- BG as the reference, each tick from the calibrated value is a 10.156 mV delta, as discussed previously. Note that the BG still has a 1% tolerance.

Reading and calibrating to the reference voltage every pass through the mux eliminates any drift. The voltage can be from a precision external reference such as an LM4040.

Adding I2C

Now that the PSoC is operating as a voltage sequencer/monitor, the overall system can be enhanced by allowing an external device to read the actual voltage levels. The PSoC supports a variety of interfaces to do this task, including the popular I2C bus.

OBSOLETE

In the PSoC Designer user module library, there are two choices for adding an I2C interface. There is the I2CHW User Module and the EZI2C User Module.

The I2CHW User Module provides the code and hardware support to allow reading of and writing to different buffer areas. It relies on the application code to manage some of the control of these buffers and therefore, is an excellent User Module for those applications where unique I2C protocols need to be supported.

The EZI2C User Module is a little different than the I2CHW User Module in that it is easier to use when reading and writing buffer areas. It also has dynamic I2C addressing built in.

Both user modules provide a solid I2C interface and it is really a matter of personal preference as to which one you use. If reading and writing memory arrays is the task at hand, the EZI2C is recommended because it is much easier to set up and use.

For this voltage sequencer/monitor design, there are eight voltage inputs that we are monitoring. If we want to provide these eight voltage values to an external system over I2C, the EZI2C User Module is a good choice.

Because we already have the eight voltage levels in eight different variables, all that we need to do is provide access to the eight different values. Figure 5 shows the declared C variables for the eight different voltages. Without the EZI2C, these declarations were simply eight unsigned char definitions. However, because we want to read these variables in order over the I2C bus, we need to ensure that they are sequential in memory. Most often, the C compiler will have them in order in memory and we could have read them without any modifications, however, we will add a structure to ensure that all goes well.

Figure 5. Declaring a Structure

```
struct Status {
  unsigned char Volt_1;
  unsigned char Volt_2;
  unsigned char Volt_3;
  unsigned char Volt_4;
  unsigned char Volt_5;
  unsigned char Volt_6;
  unsigned char Volt_7;
  unsigned char Volt_8;
} MyStatus;
```

A structure in C is a way to organize complex data. In this case, the data is not very complex but it is convenient to use and to “package” the data into one entity. We can add other variables inside the structure and still have one “package” that represents the voltage monitor status.

Again referring to Figure 5, we simply add the keywords `struct Status {`, where `struct` indicates the following is a structure and `Status` is an optional tag that is a shorthand label we can use to refer to this structure. We add the keyword `}MyStatus,` where `MyStatus` is a structure of the type `Status`.

Instead of including `MyStatus` at the end, we could have declared,

```
structure Status MyStatus;
```

which says `MyStatus` follows the structure as defined by the `Status` type.

Now that we have the variables “packaged“ in a neat container, we can tell the EZI2C where that container is and how large it is.

Figure 6. EZI2C Initialization Code

```
// EZI2C Initialization

// Set buffer and start user module

EzI2Cs_1_SetRamBuffer(sizeof(MyStatus), 0,
(BYTE *) &MyStatus);

EzI2Cs_1_Start();
```

The EZI2C initialization code is shown in Figure 6. As you can see, it is very easy to set up and run. The first state of SetRamBuffer has three parameters: buffer size, allowed bytes to write, and the pointer to the buffer.

In the statement, we used the function `sizeof` to determine the size of the buffer. This is handy in that we can change the buffer size and not worry about updating this parameter. We could set this value to 8 since that is the size of our buffer.

The second parameter is the size of the writable area, starting with the first location. Since we want these voltage values to be read only, we set this number to 0. If we added control variables that we wanted to write, we would place them first in the structure and specify how many write variables there are.

The last parameter is simply a pointer to the structure or other memory array.

This set-up results in an I2C memory map that’s accessible from an external I2C master.

Figure 5. I2C Register Map

Register	Contents
00	Volt_1 ADC Value
01	Volt_2 ADC Value
02	Volt_3 ADC Value
03	Volt_4 ADC Value
04	Volt_5 ADC Value
05	Volt_6 ADC Value
06	Volt_7 ADC Value
07	Volt_8 ADC Value

Refer to the EZI2C User Module datasheet in PSoC Designer for additional details on the EZI2C function as well as bus communication illustrations.

That is all there is to adding I2C to the voltage monitor project. You declare the array or structure, initialize the EZI2C engine, and the EZI2C firmware controls all the I2C transactions without application code intervention.

OBSOLETE

Large Memory Model

The code example with this appnote includes support for the Large Memory Model (LMM) PSoCs. Additional information about the LMM can be found in application note AN2218. In a PSoC that has the LMM, variables in an ISR prefer to be in page 0. Therefore, any variables that we define in C that will be updated in the ADC ISR, we want to force to be in page 0. The comments in the example code highlight these areas. More information regarding placing variables into specific memory pages can be found in application note AN3989.

Other Options

This design example has illustrated a system where the voltage values control only the output enables. It also has shown how to add an I2C interface to read the voltage values from the rails. This system can easily be expanded to provide a more capable device. Another approach is to create alarms that trip when the voltage is low or high and send back the alarm information to the service processor. Either way, the data is there and easily put into a format that best suits the system's needs.

The voltage sequencer in this project uses a single value as the trip level. However, it may be beneficial to add hysteresis. If this is desired, simply modify the 'if' statements that check the minimum level of the voltage to be higher when to enable and lower to indicate a voltage loss.

Another function that can be added is a voltage-margining control. Some systems require voltages to be increased and decreased though a range to test how well the design operates in over voltage and under voltage situations. Since the PSoC is already monitoring voltages, a function can be added to control the output of the voltage regulators. Depending on the power supply, the interface to control the output voltage may be an analog control, such as a DAC, or a serial bus, such as I2C or SMBus. Either way, the PSoC has the capability to control and adjust to meet the necessary output levels.

Summary

The PSoC device can be easily configured to monitor up to eight different voltages. It can also supply a variety of enables and status lines that allow tailoring to any system need. Expensive linear devices typically used for voltage monitoring can be replaced with a low-cost PSoC (and you get an MCU for free).

About the Author

Name:	Ernie Buterbaugh
Title:	Field Applications Engineer, Member of the Technical Staff
Background:	BSEE from Pennsylvania State University. More than 25 years experience with embedded processors, board level design, ASIC, FPGA and CPLD design. Has authored a variety of Application Notes, articles, and the book: *Perfect Timing: A Design Guide for Clock Generation and Distribution*
Contact:	ewb@cypress.com

OBSOLETE

Document History

Document Title: PSoC® 1 Power Management – Voltage Monitoring and Sequencing with PSoC and I2C - AN2379

Document Number: 001-17215

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1334605	HMT	07/31/2007	Enter App Note into spec system. Update template.
*A	1778605	EWB	11/28/2007	Obtained spec. # for note to be added to spec. system.
*B	3180507	GDN	02/23/2011	Updated the content and code example as per WWP NPS improvement
*C	3236538	EWB	04/21/2011	Updated Title.
*D	4406340	PRKU	06/13/2014	Obsolete Spec. Information is available in AN78646

OBSOLETE

PSoC is a registered trademark of Cypress Semiconductor Corp. "Programmable System-on-Chip," PSoC Designer, and PSoC Express are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

In March of 2007, Cypress recataloged all of its Application Notes using a new documentation number and revision code. This new documentation number and revision code (001-17215, beginning with rev. **), located in the footer of the document, will be used in all subsequent revisions.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/

© Cypress Semiconductor Corporation, 2006-2011. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.