



## Application Note

AN2339

### ***Storing Calibration Factors into Flash Memory Within a PSoC Express™ Application***

**Author:** Dave Funston

**Associated Project:** Yes

**Associated Part Family:** CY8C21xxx, CY8C24x23A, CY8C24x94, CY8C27xxx, CY8C29xxx

**PSoC Express Version:** 2.1

**Associated Application Notes:** AN2015

#### **Abstract**

This Application Note demonstrates how to facilitate the storage of calibration factors (or other “relatively constant” data) in non-volatile memory (Flash) within a PSoC Express application. It uses a simple 1-value example with architecture that can be extended to store multiple values (up to 64 bytes total) in Flash. The note also demonstrates use of a state machine valuator transfer function to create two different operation modes for the PSoC Express application.

#### **Introduction**

Many microcontroller applications require the use of calibration factors in order to operate correctly. System or component calibrations can occur in the manufacturing process, during installation, or in field usage. For a variety of reasons, storing calibration data into non-volatile memory is highly desirable. The Flash memory on a PSoC® mixed-signal controller can meet this need, but PSoC Express 2.0's graphical programming constructs cannot directly access the Flash memory.

This Application Note discusses PSoC Express' customization mechanism, which will be used to access the Flash memory, and it reiterates important concepts from AN2015 "Flash APIs" for using Flash memory. The example application demonstrates software architecture for directly addressing bytes or words in Flash memory (to emulate an EEPROM) and an interlock mechanism for preventing undesired system behavior while gathering and storing data.

#### **PSoC Express Customization**

When PSoC Express builds an application, it generates a directory containing a set of 'C' and assembly language source code and header files inside the Express project directory, and it compiles and links these files into binary code. With two exceptions, this entire set of files is deleted and regenerated each time PSoC Express builds the application. PSoC Express preserves the contents of two files (*custom.c* and *custom.h*) whose purpose is to integrate features not available through PSoC Express' graphical elements. This mechanism enables the programmer to use PSoC Express to develop future application revisions without the complications of manually post processing the application project with PSoC Designer™ prior to creating the binary code image.

The default copy of *custom.c* contains empty code stubs for three functions called by main() that take no parameters and return nothing, while *custom.h* contains their prototypes. The function names are:

- CustomInit()
- CustomPostInputUpdate()
- CustomPreOutputUpdate()

`CustomInit()` is executed once after all drivers and valuator have been initialized. `CustomPostInputUpdate()` and `CustomPreOutputUpdate()` are each called once per iteration of the control loop. `CustomPostInputUpdate()` is called immediately after the input data has been gathered from the hardware, and `CustomPreOutputUpdate()` is called immediately before the output data is sent to the hardware.

This architecture enables the user to perform custom initializations and custom processing of the incoming and outgoing data at appropriate times in the control loop simply by adding code inside these functions and adding supporting functions inside `custom.c`. After these modifications are done, the programmer can rebuild the application, and the custom code is included in the resulting binary code image.

For this example, code is added to `CustomInit()` to retrieve a calibration factor from the Flash memory, and code is added to `CustomPreOutputUpdate()` to write the calibration factor into the Flash memory when required.

## General Flash Considerations

There are some considerations to keep in mind when using the PSoC Flash memory.

First, PSoC Flash memory can be read in 1-byte increments, but it must be written in 64-byte blocks. Therefore, the application should reserve Flash in 64-byte increments and create a 64-byte buffer in RAM to facilitate preservation of Flash values that should not be changed.

Second, the nominal Flash write cycle time is 10 ms/block. In cold temperatures, it can be as long as 20 ms/block. For each block modification, this cycle must be executed once for erasing the block and once for writing the new data, so any modification of Flash memory will take a minimum of 20 ms/block. In some applications, this could be detrimental to proper operation of control functions, and in such cases, the controls must be interlocked for safety. The example application interlocks its control output.

Last, when writing to Flash memory, knowledge of the chip temperature is critical to meeting the write cycle life specification of the PSoC Flash. The PSoC chip contains an internal temperature sensor, but PSoC Express does not have an input driver for it. However, this limitation can be ignored if the chip temperature is held within certain limits.

AN2015 indicates that for operation in the 0° - 85°C range, a constant temperature value may be input to the Flash writing function if the actual chip temperature differs from that value by less than 25°C. The example application assumes a constant temperature of 25°C. This assumption enables use of the Flash writing capability between 0°C and 50°C actual temperature without degrading the write cycle life.

## Application Overview

The example application for this note contains several components that represent conceptual elements common to many controls applications:

- Sensor input (Voltage)
- Calibration factor (Threshold)
- Control output (VoltageLight)
- User inputs (Button1 and Button2)
- Different operation modes (normal and calibration)
- Status/mode indicator (ModeLight)
- Interlock to prevent undesired behavior during calibration (VoltageLight turned off while in calibration mode)

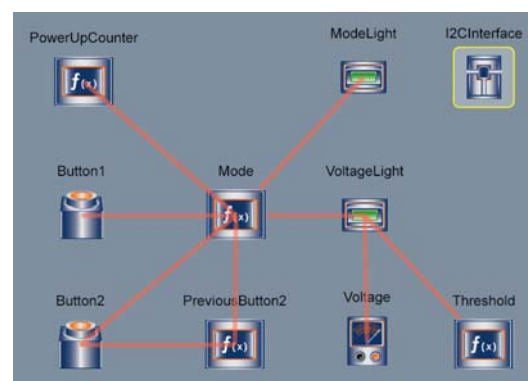


Figure 1. Example Application

In normal operation, the application measures a voltage (Voltage), and compares it to a threshold voltage whose value (Threshold) has been retrieved from non-volatile memory during initialization of the program. If Voltage exceeds Threshold, an LED (VoltageLight) blinks. If Voltage is less than Threshold, VoltageLight is held on.

In order to get into calibration mode, the user must simultaneously hold down two buttons (Button1 and Button2) while powering up or resetting the PSoC. In calibration mode, Voltage is read as the source for calibrating Threshold. When the user pushes Button2, Voltage's value is stored in the Threshold valuator and the Flash memory. Also, VoltageLight is kept off while in calibration mode.

The user must cycle power or reset the PSoC to exit calibration mode and return to normal mode.

A second LED, (ModeLight) indicates whether the PSoC is operating in normal mode or calibration mode. For normal mode, ModeLight stays on, and for calibration mode, ModeLight blinks.

The example application also includes an optional object, I2CInterface, which enables monitoring of driver and valuator values on the CY3210-PSoCEval1 through an I2C interface.

## Operation Mode Implementation

One of the main tasks to accomplish in PSoC Express is to make the application operate in two different modes. This is accomplished through the use of the mode valuator, which contains a state machine transfer function. The state machine initializes to the PowerUp state, and remains in that state for 16 control loop iterations (1/4 second at 64 Hz execution rate). PowerUpCounter is used as a one-shot counter to track the number of iterations following application initialization. PSoC Express initializes it to 0, and its transfer function is a priority encoder with two lines:

- o if PowerUpCounter<128 then  
PowerUpCounter + 1
- o else if 1 then 128

This priority encoder causes the valuator to increment itself once per control loop iteration from 0 until it reaches 128 (2 seconds at 64 Hz execution rate) where it stays until reset or power-down. Allowing PowerUpCounter to go up to 128 gives the application programmer some flexibility in selecting the number of iterations to execute before changing state.

After the 16 control loop iterations, the state machine goes into the Calibration state (which represents calibration mode) if both Button1 and Button2 are on. Otherwise, the state machine transitions into the Normal state (which represents normal mode). After the transition into either the Normal state or the Calibration state, the state machine remains in that state. The Calibration state has a transition back to itself whose purpose is to occur when the user presses Button2. This event is captured by checking that Button2 is on and the previous value of Button2 (represented by the loop delay transfer function valuator PreviousButton2) is off. This transition named, SaveCalFactors, represents a trigger for the application to store data in the Flash memory.

Other valuator and output drivers use this simple state machine to direct their operation. For instance, ModeLight uses a table lookup transfer function using the value of the state to determine whether it is on (for the Normal state), off (for the PowerUp state), or blinking (for the Calibration state). VoltageLight uses a priority encoder with the following lines judging the value of the state to determine whether the LED should be off (interlocked) or should direct its operations based on the Voltage value (normal operation).

- o if Mode\_state == Mode\_state\_\_PowerUp  
then VoltageLight\_\_OFF
- o else if Mode\_state ==  
Mode\_state\_\_Calibration then  
VoltageLight\_\_OFF
- o else if Voltage > Threshold then  
VoltageLight\_\_BLINKING
- o else if 1 then VoltageLight\_\_ON

Also, the code in CustomPreOutputUpdate() will check for the occurrence of the SaveCalFactors transition to determine whether to write the calibration data to the Flash memory.

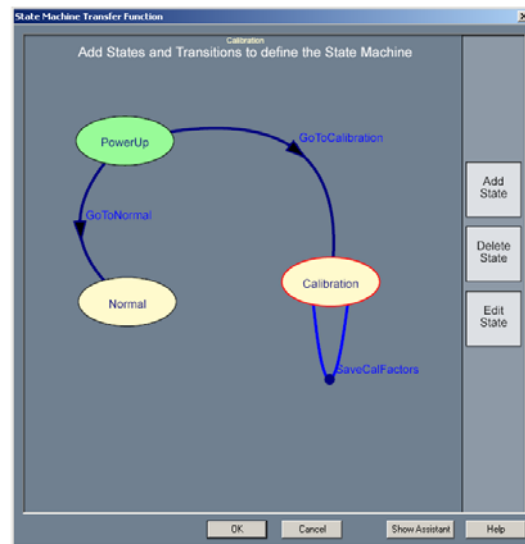


Figure 2. Mode State Machine

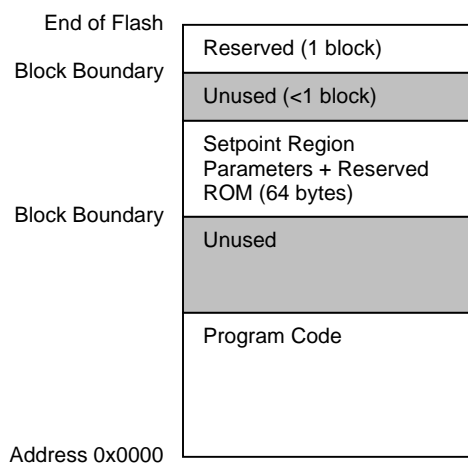
If the developer programming the application chooses, he could add a transition from Calibration state to Normal state (and/or the other direction) by way of some other event(s).

## Calibration Factor Implementation

Threshold is the valuator containing the calibration factor. Its transfer function is a priority encoder with one line: `if 0 then 0`. Since the `if` clause evaluates to 0, the `then` clause never executes. Therefore, Threshold's value remains constant unless changed by code within *custom.c*. Threshold's data type is set to continuous to match Voltage's data type.

## Flash Memory Interface Implementation

This example uses the FlashBlock API within *custom.c*. The FlashBlock API is part of the standard PSoC library and described in detail in AN2015. The system header file *FlashBlock.h* contains the data structure definitions and interface function prototypes.



**Figure 3. PSoC Express Flash Memory Map (64 bytes Reserved ROM)**

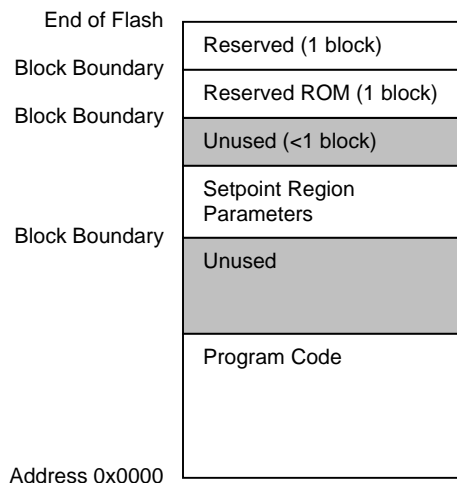
An area of Flash memory must be allocated for storing the calibration factor. PSoC Express allows the user to reserve Flash memory for program use through the Reserved ROM option available in the build dialog. Figure 3 shows a Flash memory map as allocated by PSoC Express.

Keeping the Flash blocks used by the custom code separate from the blocks allocated for other uses by PSoC Express is desirable for robustness (but not for reduced memory usage). However, because PSoC Express combines the Flash allocation for Reserved ROM with the allocation for Setpoint Region setpoints and hysteresis, the user must take two steps to ensure that the Reserved ROM is separate from the block(s) containing the Setpoint Region parameters.

First, the Reserved ROM option must be set in 64-byte (1 block) increments, and second, the user must “shift” the Reserved ROM so that it covers the unused area below PSoC Express' internally reserved block. This shift is accomplished by identifying an appropriate start address to assign to the start of Reserved ROM, and programmatically accessing the Flash area between this start address and the start address of PSoC Express' internally reserved block.

The example program uses less than 64 bytes of Flash memory for data storage, so the Reserved ROM option is set to 64 bytes, and the Flash Interface option is set to Disable when the program builds. In addition to reserving Flash area for Reserved ROM, building the program creates the source code directory structure, *custom.c*, and *custom.h*.

The next step is to determine the Reserved ROM's start address. This address can be obtained by subtracting the size of Reserved ROM (64 bytes for the example) from the start address of PSoC Express' internally reserved area, which is found in the PSoC Express generated file, *Calibration.c*. The `#pragma abs_address` directive at the beginning of this file defines the start address of PSoC Express' internally reserved area. For this example, which was built for a CY8C29466, the internally reserved start address is 0x7FC0, so the Reserved ROM start address is 0x7F80. Dividing this address by 64 (0x40), results in the block ID (0x7F80/0x40 = 0x1FE), which is used to address Flash memory via the FlashBlock API. Figure 4 shows how the example program re-maps the Flash memory.



**Figure 4. Example Program Flash Memory Map (64 bytes Reserved ROM)**

With the block ID determined, modifications to *custom.c* can be made. At the top of *custom.c*, the `#include <FlashBlock.h>` directive exposes the FlashBlock API function prototypes and data type definitions. The `#include "CMXSystemExtern.h"` directive exposes the SystemVars global variable structure that houses the values of the valuator and input and output drivers defined in the PSoC Express application. Last, the `#include "FunctionParamDecl.h"` directive exposes the constant value names defined for discrete type input drivers, output drivers, and valuator such as the LEDs, the buttons, and the mode state machine.

Since the block ID of the reserved Flash area will be used multiple times, the `#define FLASH_BLOCK_ID 0x1FE` directive exists to address the need for this constant. Also, the `#define NO 0` and `#define YES 1` directives have been added to create values for a flag that indicates the necessity for a Flash memory write operation. These three directives are inside *custom.c* in order to prevent possible conflicts with constants or variables of the same name in the other modules auto-generated by PSoC Express.

A 'C' union serves as a RAM buffer for the Flash block. This buffer contains a 64-byte array (`FlashBlockBuffer`) that serves as the raw block buffer. This block buffer is overlaid with a data structure (`Vars`) that defines what the values at each location represent. When a future revision of the application requires more values to be stored in non-volatile memory, fields can be added to the `Vars` structure to accommodate these new values. The size of `Vars` must not exceed 64 bytes.

Now that the data structure is in place, the program must read the appropriate data out of Flash when it starts running in order to initialize the value of Threshold. This is done in the `CustomInit()` function. `CustomInit()` sets up a parameter structure containing the block ID to read, the number of bytes to read, and a pointer to the raw block buffer into which the data is written. Then, it calls `FlashReadBlock()` to put the data in the buffer. After the RAM buffer is written, the Threshold field in the `Vars` structure is copied to its respective valuator field in the SystemVars global data structure. This code is easily extended to accommodate additional Flash data use by adding the necessary statements to copy the new values from the `Vars` structure to the SystemVars structure.

The last issue of concern is writing data into the Flash memory, and `CustomPreOutputUpdate()` addresses this need. Because Flash memory has a limited write cycle life, and because the value of Threshold is not intended to change often, `CustomPreOutputUpdate()` writes to Flash only if the user wishes to save the value of Threshold, and the new value differs from the old value. To do this, `CustomPreOutputUpdate()` first looks to see if the SaveCalFactors transition in the mode state machine has occurred. The occurrence of this event indicates the user wishes to save the new data. When this condition is satisfied, the `SaveRequired` flag is initialized to NO, and Threshold is compared with its desired new value.

If the current value differs from the new value, the new value is saved in the valuator, the new valuator is copied into its respective field in the `Vars` structure of the Flash buffer, and the `SaveRequired` flag is set to YES. After the `if(...){...}` block performing this work is executed, `SaveRequired` is checked. If its value is YES, the buffer is written to the Flash memory.

When future revisions of the application require additional Flash data usage, one `if(...){...}` code block must be added to `CustomPreOutputUpdate()` for each Flash value. The block checks for a change in the desired valuator value. If the change occurred, the code copies the new value into the valuator and its respective field in the Flash buffer, and the code flags that a write to Flash is required.

Example code is in Appendices 1, 2, 3, 4, and 5. Appendix 1 has the auto-generated *Calibration.c* for reference. Appendix 2 has the auto-generated *SystemVars.h*, which shows the structure of the SystemVars that is exposed to *custom.c* by *CMXSystemExtern.h* through the statement `extern SYSTEM_VARS_STRUC SystemVars`. Appendix 3 has *custom.h*, which is the default copy as no modifications to this file are required for this example. Appendix 4 shows the entire contents of *custom.c* after revision to enable Flash memory reading and writing. An example copy of *custom.c* provided with this Application Note can be used to replace the default copy that PSoC Express generates. Last, Appendix 5 shows an example of *custom.c* if an additional value (Threshold1) is used in the Flash memory, and its value depends on an additional input (Voltage1).

## Conclusion

This Application Note describes architecture for enabling a PSoC Express application to read and write Flash memory. This architecture uses a 'C' union containing a 64-byte block buffer overlaid with a data structure to allow routines in *custom.c* to directly address data within the buffer. These routines read and write the buffer to Flash memory, and interface with the graphical elements of the PSoC Express application. This architecture is limited to working with a maximum of 64 bytes of Flash memory, and it uses more than the optimum amount of Flash memory in order to ensure that Flash areas allocated for other uses by PSoC Express are not overwritten. Reducing the amount of Flash memory used can be accomplished by overlapping the blocks used for Reserved ROM with the blocks used for Setpoint Region parameters, but the details of implementation are outside the scope of this document.

In addition to showing how to access the Flash memory, this note gives an example of using a state machine to launch different operation modes for the PSoC Express application.

## Appendix 1: Calibration.c

```

//*****
//*****
// FILENAME: calibration.c
// @Version@
// `@PSOC_VERSION`
//
// DESCRIPTION: This files contains the calibration constants for the
//              ADC. Currently these values are default values that
//              are not calibrated.
//
//-----
//              Copyright (c) Cypress MicroSystems 2004. All Rights Reserved.
//*****
//*****

#pragma abs_address:0x7FC0
const int CountsPerVolt = 25206; // ADC gain for 0 to 2.6 volt range.
const int ADC_Offset    = 0;     // ADC offset in counts

// This array of offsets allows for custom calibration
// of each input that uses the mVolts channel. The offset
// will be in the drivers native units. For the mVolts
// driver it will be in mVolts. For a temperature driver
// it will be in tenths of degrees, etc.
const int imVolts_Chan_Offset[8] = {0,0,0,0,0,0,0,0};
#pragma end_abs_address

```

## Appendix 2: SystemVars.h

```

//
// SystemVars.h
//

#ifndef SYSTEM_VARS_H
#define SYSTEM_VARS_H

#include "CMXSystem.h"
#include "driverdecl.h"

void UpdateVariables( void);

typedef struct
{
    struct
    {
        BYTE pse_Button1;
        BYTE pse_Button2;
        int  pse_Voltage;
        int  pse_Threshold;
        BYTE pse_ModeLight;
        BYTE pse_VoltageLight;
        BYTE pse_Mode_state;
        BYTE pse_Mode_transition;
        BYTE pse_PowerUpCounter;
        BYTE pse_PreviousButton2;
    } ReadOnlyVars;
} SYSTEM_VARS_STRUC;

#define SYSTEM_VARIABLE_COUNT 12 // Total size of all variables. ( 2 x ints + 1 x BYTES )
#define SYSTEM_RW_VARIABLE_COUNT 0 // Total size of the read/write variables. ( 2 x ints + 1 x BYTES )

#endif

```

## Appendix 3: custom.h

```
//
// Custom.h
//
// Prototypes for custom functions, including fixed functions
//
#include <m8c.h>

extern void CustomInit( void);
extern void CustomPostInputUpdate( void);
extern void CustomPreOutputUpdate( void);
```

## Appendix 4: custom.c

```
//
// Custom.c
//
// Custom functions for insertion into PSoC Express projects, including fixed functions
//

#include "custom.h"
#include "CMXSystemExtern.h"
#include "functionparamdecl.h"
#include <FlashBlock.h>

// Flash Block ID
// PSoC Express reserved area start address 0x7FC0 is from calibration.c
// Available area for Reserved ROM is below this address.
// 1 block (0x40 bytes) is used. -> Reserved ROM start address 0x7F80 = 0x7FC0 - 0x0040
// Block ID = 0x1FE = 0x7F80/0x40
#define FLASH_BLOCK_ID 0x1FE

#define NO 0
#define YES 1

union FLASH_BUFFER_UNION
{
    BYTE FlashBlockBuffer[64];
    struct E2PROM_VARS
    {
        int Threshold;
    } Vars;
};

// Buffer for interfacing with the Flash emulated E2PROM. This buffer is visible
// only in this module.
union FLASH_BUFFER_UNION FlashBuffer;

void CustomInit()
{
    // START OF CustomInit()
    FLASH_READ_STRUCT FlashReadRecord;

    FlashReadRecord.wARG_BlockId = FLASH_BLOCK_ID;
    FlashReadRecord.pARG_FlashBuffer = FlashBuffer.FlashBlockBuffer;
    FlashReadRecord.wARG_ReadCount = 64;

    // Read in the contents of the Flash emulated E2PROM.
    FlashReadBlock(&FlashReadRecord);

    // Copy each parameter from the buffer to its global variable location
    SystemVars.ReadOnlyVars.pse_Threshold = FlashBuffer.Vars.Threshold;

    // END OF CustomInit()
}

void CustomPostInputUpdate()
{
    // START OF CustomPostInputUpdate()

    // END OF CustomPostInputUpdate()
}

void CustomPreOutputUpdate()
{

```

```
// START OF CustomPreOutputUpdate()
FLASH_WRITE_STRUCT FlashWriteRecord;

char SaveRequired;

// Check to see if the user desires to save new calibration factors
if (SystemVars.ReadOnlyVars.pse_Mode_transition == pse_Mode_transition__SaveCalFactors)
{
    // Initialize the SaveRequired flag to "not required"
    SaveRequired = NO;

    // Check each calibration factor for a change in value. If the
    // value has changed then, save the factor in RAM, save the factor in the
    // Flash buffer, and flag the need to save the factor to Flash
    if (SystemVars.ReadOnlyVars.pse_Threshold != SystemVars.ReadOnlyVars.pse_Voltage)
    {
        SystemVars.ReadOnlyVars.pse_Threshold = SystemVars.ReadOnlyVars.pse_Voltage;
        FlashBuffer.Vars.Threshold = SystemVars.ReadOnlyVars.pse_Threshold;
        SaveRequired = YES;
    }

    // Check to see if a Flash write is required, and
    // perform the write if so
    if (SaveRequired == YES)
    {
        // Prep the Flash writing function
        FlashWriteRecord.wARG_BlockId = FLASH_BLOCK_ID;
        FlashWriteRecord.pARG_FlashBuffer = FlashBuffer.FlashBlockBuffer;
        FlashWriteRecord.cARG_Temperature = 25;

        // Write to Flash
        bFlashWriteBlock(&FlashWriteRecord);
    }
}

// END OF CustomPreOutputUpdate()
}
```

## Appendix 5: Example custom.c with Multiple Flash Variables

```

//
// Custom.c
//
// Custom functions for insertion into PSoC Express projects, including fixed functions
//

#include "custom.h"
#include "CMXSystemExtern.h"
#include "functionparamdecl.h"
#include <FlashBlock.h>

// Flash Block ID
// PSoC Express reserved area start address 0x7FC0 is from calibration.c
// Available area for Reserved ROM is below this address.
// 1 block (0x40 bytes) is used. -> Reserved ROM start address 0x7F80 = 0x7FC0 - 0x0040
// Block ID = 0x1FE = 0x7F80/0x40
#define FLASH_BLOCK_ID 0x1FE

#define NO 0
#define YES 1

union FLASH_BUFFER_UNION
{
    BYTE FlashBlockBuffer[64];
    struct E2PROM_VARS
    {
        int Threshold;
        int Threshold1;
    } Vars;
};

// Buffer for interfacing with the Flash emulated E2PROM. This buffer is visible
// only in this module.
union FLASH_BUFFER_UNION FlashBuffer;

void CustomInit()
{
    // START OF CustomInit()
    FLASH_READ_STRUCT FlashReadRecord;

    FlashReadRecord.wARG_BlockId = FLASH_BLOCK_ID;
    FlashReadRecord.pARG_FlashBuffer = FlashBuffer.FlashBlockBuffer;
    FlashReadRecord.wARG_ReadCount = 64;

    // Read in the contents of the Flash emulated E2PROM.
    FlashReadBlock(&FlashReadRecord);

    // Copy each parameter from the buffer to its global variable location
    SystemVars.ReadOnlyVars.pse_Threshold = FlashBuffer.Vars.Threshold;
    SystemVars.ReadOnlyVars.pse_Threshold1 = FlashBuffer.Vars.Threshold1;

    // END OF CustomInit()
}

void CustomPostInputUpdate()
{
    // START OF CustomPostInputUpdate()

    // END OF CustomPostInputUpdate()
}

void CustomPreOutputUpdate()
{
    // START OF CustomPreOutputUpdate()
    FLASH_WRITE_STRUCT FlashWriteRecord;

    char SaveRequired;

    // Check to see if the user desires to save new calibration factors
    if (SystemVars.ReadOnlyVars.pse_Mode_transition == pse_Mode_transition__SaveCalFactors)
    {
        // Initialize the SaveRequired flag to "not required"
        SaveRequired = NO;

        // Check each calibration factor for a change in value. If the

```

```
// value has changed then, save the factor in RAM, save the factor in the
// Flash buffer, and flag the need to save the factor to Flash
if (SystemVars.ReadOnlyVars.pse_Threshold != SystemVars.ReadOnlyVars.pse_Voltage)
{
    SystemVars.ReadOnlyVars.pse_Threshold = SystemVars.ReadOnlyVars.pse_Voltage;
    FlashBuffer.Vars.Threshold = SystemVars.ReadOnlyVars.pse_Threshold;
    SaveRequired = YES;
}

if (SystemVars.ReadOnlyVars.pse_Threshold1 != SystemVars.ReadOnlyVars.pse_Voltage1)
{
    SystemVars.ReadOnlyVars.pse_Threshold1 = SystemVars.ReadOnlyVars.pse_Voltage1;
    FlashBuffer.Vars.Threshold1 = SystemVars.ReadOnlyVars.pse_Threshold1;
    SaveRequired = YES;
}

// Check to see if a Flash write is required, and
// perform the write if so
if (SaveRequired == YES)
{
    // Prep the Flash writing function
    FlashWriteRecord.wARG_BlockId = FLASH_BLOCK_ID;
    FlashWriteRecord.pARG_FlashBuffer = FlashBuffer.FlashBlockBuffer;
    FlashWriteRecord.cARG_Temperature = 25;

    // Write to Flash
    bFlashWriteBlock(&FlashWriteRecord);
}

// END OF CustomPreOutputUpdate()
}
```

---

## About the Author

**Name:** Dave Funston

**Title:** Applications Engineer, Staff

**Background:** Dave Funston supports PSoC Express at Cypress Semiconductor in Lynnwood, WA. He holds a Master's degree in Mechanical Engineering from the University of California at Davis, and has programmed microcontrollers since 1996.

**Contact:** [dfu@cypress.com](mailto:dfu@cypress.com)

---

Cypress Semiconductor  
2700 162<sup>nd</sup> Street SW, Building D  
Lynnwood, WA 98087  
Phone: 800.669.0557  
Fax: 425.787.4641

<http://www.cypress.com/>

Copyright © 2006 Cypress Semiconductor Corporation. All rights reserved.

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.

The information contained herein is subject to change without notice. Made in the U.S.A.