AN2015

PSoC® 1 – Getting Started with Flash & E2PROM

Author: Praveen Kumar M
Associated Project: Yes
Associated Part Family: CY8C20x34, CY8C20xx6, CY8C21xxx, CY8C22x45, CY8C23x33, CY8C24x23A, CY8C24x94, CY8C27x43,CY8C28xxx, CY8C29x66
Software Version: PSoC® Designer™ 5.4 SP1
Related Application Notes: AN44168

AN2015 enables the reader to get started with the flash memory in PSoC® 1 by focusing on PSoC 1 flash architecture Read, Write algorithm, Protection modes and their impact on user applications. The example projects demonstrates the two methods for reading and writing to the flash memory within a user application, E2PROM User Module and Flash block API library that is included in the PSoC Designer™ integrated development environment (IDE).

Contents

1 Introduction....1
2 PSoC Resources....2
2.1 PSoC Designer....2
2.2 Code Examples....3
2.3 Technical Support....4
3 Flash Architecture....5
3.1 Flash Write Algorithm....6
3.2 Selecting the Appropriate Temperature....6
3.3 Flash Protection....7
4 How to Write to Flash....8
4.1 E2PROM User Module....8
4.2 Flashblock API Library....9
5 Flashblock API Description....11
6 Example Projects....12
6.1 Hardware Setup / Demo....13
6.2 Example Project 1: E2PROM – CY8C29466-24PXI....14
6.3 Example Project 2: Flashblock – CY8C29466-24PXI....16
6.4 Example Project 3: Flashblock – CY8C27443-24PXI....18
6.5 Reading Data from Flash Using PSoC Programmer....18
7 Special Considerations....20
7.1 Interrupts and Timing....20
7.2 Voltage Stability....20
7.3 Placing Data in Flash....21
8 Summary....22
9 References....22
Document History....23
Worldwide Sales and Design Support....24

1 Introduction

Flash operations executed by user firmware are a key element of many embedded designs. The ability for a device to update its own flash is useful in many applications including bootloaders and applications that need to store nonvolatile information such as calibration data.

PSoC 1 devices provide the capability to easily read from and write to flash with either the E2PROM user module or a Flashblock API library. Although the E2PROM user module is very briefly discussed, this application note focuses on the Flashblock API library. The Flashblock API library is described in detail, demonstrating how to use the APIs.

In addition to the discussion of the Flashblock API library, an overview of the flash architecture, flash protection settings, and flash write algorithm are provided in this application note. These elements, plus additional useful design tips, are provided to help users easily design their project using flash writes and reads.

2 PSoC Resources

Cypress provides a wealth of data at www.cypress.com to help you to select the right PSoC device for your design, and quickly and effectively integrate the device into your design. In this document, PSoC refers to the PSoC 1 family of devices. To learn more about PSoC 1, refer to the application note AN75320 - *Getting Started with PSoC 1*.

The following is an abbreviated list for PSoC 1:

- **Overview:** PSoC Portfolio, PSoC Roadmap
- **Product Selectors:** PSoC 1, PSoC 3, PSoC 4, or PSoC 5LP. In addition, PSoC Designer includes a device selection tool.
- **Datasheets:** Describe and provide electrical specifications for the PSoC 1 device family.
- **Application Notes and Code Examples:** Cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples.
- **Technical Reference Manuals (TRM):** Provide detailed descriptions of the internal architecture of the PSoC 1 devices.
- **Development Kits:**
  - CY3215A-DK In-Circuit Emulation Lite Development Kit includes an in-circuit emulator (ICE). While the ICE-Cube is primarily used to debug PSoC 1 devices, it can also program PSoC 1 devices using ISSP.
  - CY3210-PSOCEVAL1 Kit enables you to evaluate and experiment Cypress's PSoC 1 programmable system-on-chip design methodology and architecture.
  - CY8CKIT-001 is a common development platform for all PSoC family devices.
- The MiniProg1 and MiniProg3 devices provide an interface for flash programming.

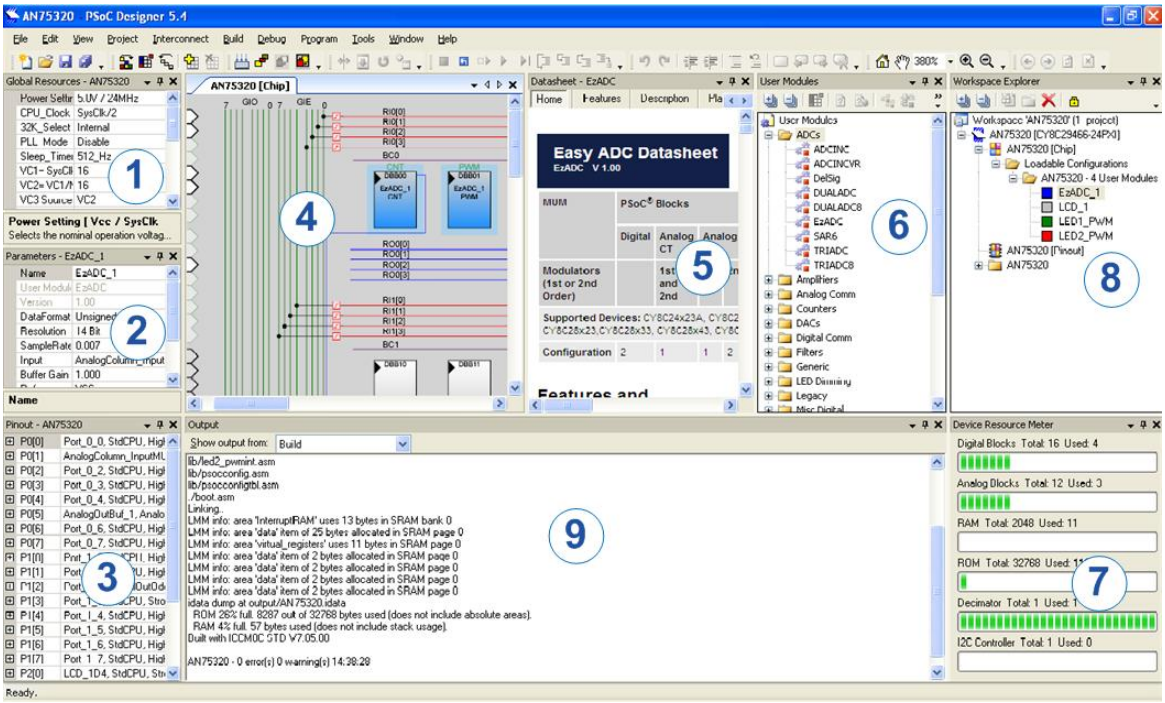
2.1 PSoC Designer

PSoC Designer is a free Windows-based Integrated Design Environment (IDE). Develop your applications using a library of pre-characterized analog and digital peripherals in a drag-and-drop design environment. Then, customize your design leveraging the dynamically generated API libraries of code. Figure 1 shows PSoC Designer windows. **Note:** This is not the default view.

1. **Global Resources –** all device hardware settings.
2. **Parameters –** the parameters of the currently selected User Modules.
3. **Pinout –** information related to device pins.
4. **Chip-Level Editor –** a diagram of the resources available on the selected chip.
5. **Datasheet –** the datasheet for the currently selected UM
6. **User Modules –** all available User Modules for the selected device.
7. **Device Resource Meter –** device resource usage for the current project configuration.
8. **Workspace –** a tree level diagram of files associated with the project.
9. **Output –** output from project build and debug operations.

Note: For detailed information on PSoC Designer, go to **PSoC® Designer > Help > Documentation > Designer Specific Documents > IDE User Guide**.

Figure 1. PSoC Designer Layout



2.2 Code Examples

The following webpage lists the PSoC Designer based Code Examples. These Code Examples can speed up your design process by starting you off with a complete design, instead of a blank page and also show how PSoC Designer User modules can be used for various applications.

http://www.cypress.com/go/PSoC1Code Examples

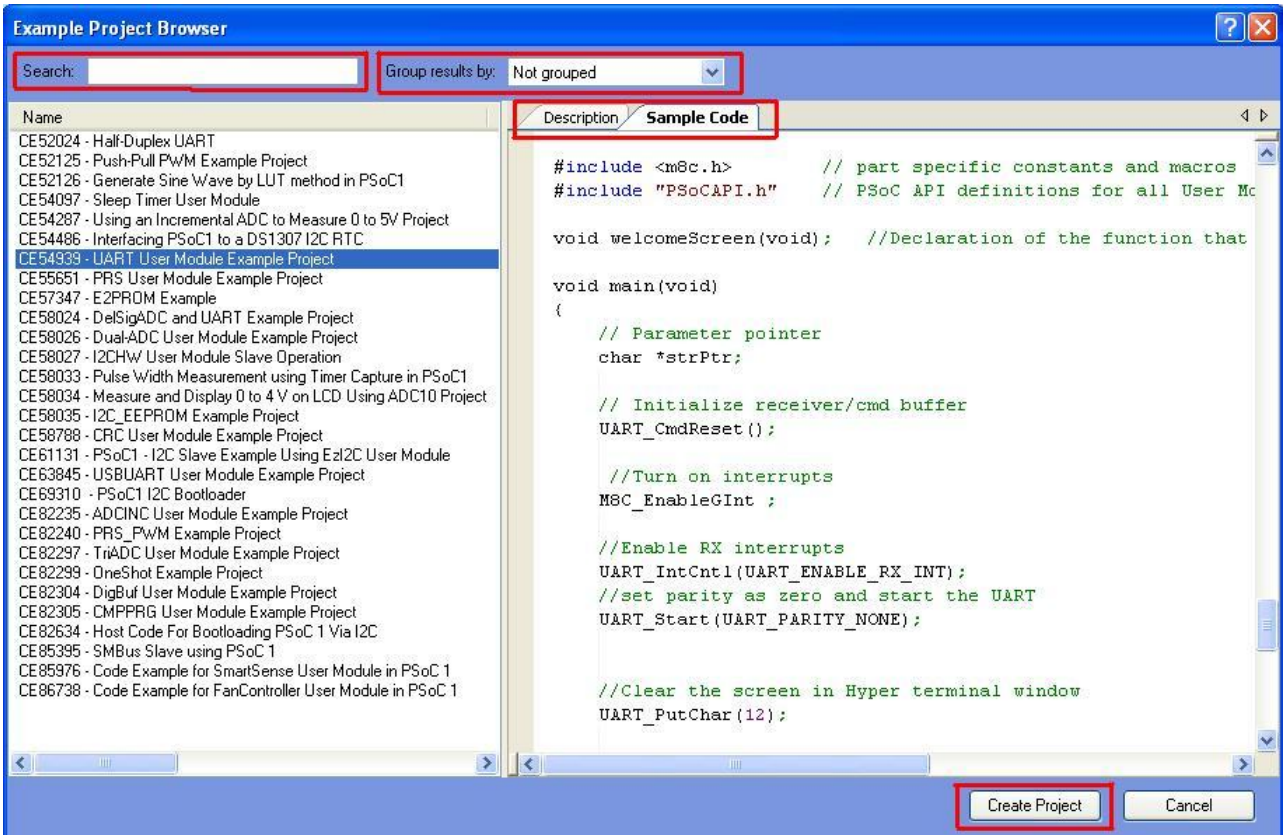
To access the Code Examples integrated with PSoC Designer, follow the path **Start Page > Design Catalog > Launch Example Browser** as shown in Figure 2.

Figure 2. Code Examples in PSoC Designer

In the Example Projects Browser shown in Figure 3, you have the following options.

- Keyword search to filter the projects.
- Listing the projects based on Category.
- Review the datasheet for the selection (on the Description tab).
- Review the code example for the selection. You can copy and paste code from this window to your project, which can help speed up code development, or
- Create a new project (and a new workspace if needed) based on the selection. This can speed up your design process by starting you off with a complete, basic design. You can then adapt that design to your application.

Figure 3. Code Example Projects, with Sample Codes



2.3 Technical Support

If you have any questions, our technical support team is happy to assist you. You can create a support request on the Cypress Technical Support page.

You can also use the following support resources if you need quick assistance.

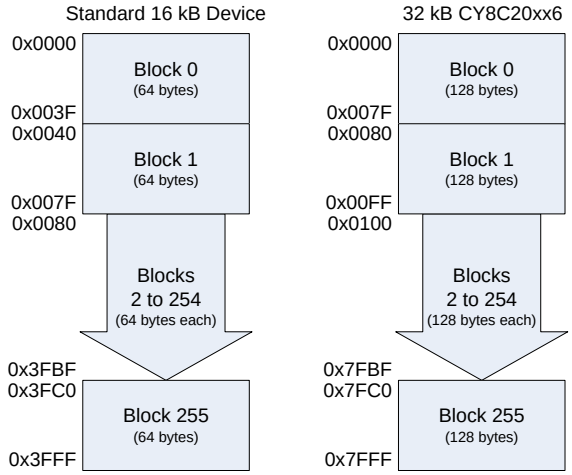
- Self-help
- Local Sales Office Locations

3 Flash Architecture

The flash within most PSoC 1 devices are organized in 64-byte blocks. The 4-kilobyte families have 64 blocks numbered 0 through 63. The 8-kilobyte families have 128 blocks numbered 0 through 127. The 16-kilobyte families have 256 blocks numbered 0 through 255. Finally, 32-kilobyte families have 512 blocks numbered 0 through 511. The CY8C20xx6 devices are an exception in that they have 128-byte flash blocks. Therefore, a 32-kilobyte CY8C20xx6 device has 256 blocks numbered 0 through 255.

Figure 4 illustrates the flash layout for a standard PSoC 1 16-kilobyte device and a 32-kilobyte CY8C20xx6 device.

Figure 4. PSoC 1 Flash Architecture



User code can modify the data in these blocks. Writing to flash requires that an entire 64-byte (or 128-byte) block be written, even if only one byte is to be modified. Block 0 (and block 1 for some devices) contains the reset and interrupt vectors and the other blocks contain program code or data. By default, the ImageCraft Compiler will place program code and data starting at the lowest memory addresses after the vector table and will fill towards higher memory addresses.

This application note assumes that you are familiar with PSoC 1 device architecture and the PSoC Designer™ IDE. If you are new to PSoC 1 device, refer to AN75320 – Getting Started with PSoC1 to explore the PSoC 1 architecture and the development tools.

Table 1. Flash Blocks in PSoC 1

Product Family	Flash (KB)	Number of Blocks
CY8C21x23 / CY8C24x23A	4	64
CY8C21x34 / 45 / CY8C23x33 / CY8C24x33	8	128
CY8C22x45 / CY8C24x94 / CY8C27x43 / CY8C28xxx	16	256
CY8C29xxx	32	512

3.1 Flash Write Algorithm

The PSoC 1 M8C processor interfaces to the flash through a set of Supervisory System Call (SSC) functions that reside in Supervisory ROM (SROM). These SSC functions provide the functionality to read, write, and erase any block of flash in the PSoC (assuming the block is unprotected).

The SROM code handles the low level steps required to read, write, and erase flash. Part of this process is calculating the amount of time the flash should be pulsed in order to properly write or erase a block of flash. The pulse width is determined based on several variables: internal calibration values stored in SROM, the CPU clock rate, and temperature. Since the flash functions have access to the calibration values and use a known CPU clock, the temperature is the only value of concern for the average user.

Typically the SSC functions do not need to be called by the user directly. The Flashblock API and E2PROM user module API (both discussed later in this application note) handle the low level details for the user. For more details regarding the low-level SSC functions, refer section 3 in PSoC 1 Technical reference manual.

3.2 Selecting the Appropriate Temperature

Temperatures play a large role in the calculation of the proper flash write and erase pulse widths, which are calculated by the flash write function. Higher temperatures require a smaller duration pulse width and lower temperatures require a larger than nominal duration pulse width.

The calculations for the erase and write pulse widths are shown below in Equation 1 and Equation 2. These clock values are used by the SROM code to determine the number of clock cycles the flash should be erased and written. The M and B values are the calibration values stored in the hidden rows of flash. T is the temperature, in degree Celsius, that needs to be provided by the user. All necessary calculations are done by the flash write function; the user is only responsible for passing an appropriate temperature value.

Equation 1: Erase Pulse Width Calculation

$$CLOCK_{ERASE} = B - \frac{2M * T}{256}$$

Equation 2: Write Pulse Width Calculation

$$CLOCK_{WRITE} = B - \frac{2M * T}{64}$$

Using the appropriate temperature value in the calculation of the pulse widths is vital to ensure the flash meets the retention and endurance electrical specifications provided in the datasheet.

Table 2. Flash Electrical Specifications

Specification	Details
$Flash_{ENPB}$	Number of erase and write cycles, each flash block can withstand
$Flash_{ENT}$	The total number of endurance cycles the entire flash array can withstand
$Flash_{DR}$	Amount of time a flash cell will retain its data

Longer retention can be achieved by “refreshing” a flash block, where the existing flash block data is read out and reprogrammed into the block (which essentially resets the data retention timer).

If a device is going to be operating in a limited temperature range between 0 °C and 85 °C, the requirements on using an accurate temperature for flash writes are relaxed. Any temperature range within a 50 °C span between 0 °C and 85 °C is considered constant with respect to endurance enhancements. For example, if a device is limited to operating between 0 °C and 50 °C, then a constant temperature of 25 °C can be used for the temperature parameter. In this case, a temperature sensor is not needed.

For the full industrial range (-40 °C to +85 °C), the user must employ a temperature sensing method (such as the FlashTemp user module) and feed the result to the temperature argument before writing to flash. Flash endurance and data retention specifications may not be met if the temperature is not properly provided to the flash write algorithm. For more information on using the FlashTemp user module and its API’s, refer to the FlashTemp user module datasheet within PSoC Designer.

The CY8C20xx6 family of devices is the only exception to this guidance. These devices use internal circuitry to ensure writes to flash execute optimally to maximize flash endurance and data retention. These devices do not require a temperature parameter to be passed by the user.

3.3 Flash Protection

PSoC 1 devices have four available flash protection settings that can be configured on a block-by-block basis. The available protection modes are shown in Table 3.

Table 3. Flash Protection Settings

	Protection Level	Internal Reads	External Reads	Internal Writes	External Writes	flashsecurity.txt
Unprotected	0	Y	Y	Y	Y	'U'
Factory Upgrade	1	Y	N	Y	Y	'F'
Field Upgrade	2	Y	N	Y	N	'R'
Full Protection	3	Y	N	N	N	'W'

Full Protection: Full protection is the default protection setting used by PSoC Designer on all blocks. Full protection prevents all external reads and writes and does not allow internal writes. This is the preferred setting for any blocks that do not need to be internally updated by firmware. This provides a high level of protection against both external attacks and accidental flash corruption.

Field Upgrade: Field Upgrade protection is the next step down in protection from Full protection. It disallows external writes and reads, but allows internal writes and reads to occur. This is the safest setting for blocks that need to be updated internally by firmware. Bootloaders typically use this setting for the bootloadable portion of flash. Bootloaders requiring protection beyond what is provided by the Field Upgrade setting may want to consider encryption or some other form of protecting the bootload interface, since an external attack can still potentially occur if the bootload process is reverse engineered and initiated by an unwanted host.

Factory Upgrade: Factory Upgrade protection is rarely used, but is useful in an application where a device needs to have individual blocks updated by an external programmer. The protection setting does not allow external reads, but allows external writes, internal reads, and internal writes. This setting prevents someone from being able to directly read a block externally, but if a particular block (or set of blocks) needs to be updated by an external programmer without erasing the entire memory, this setting is ideal. Factory Upgrade protection is not recommended for designs requiring a high level of security, as there is no protection against an external attacker inserting their own code into a working system to either extract information from the device or otherwise alter the operation of the design.

Unprotected: The Unprotected setting allows all external and internal writes and reads. This protection setting provides a minimal level of protection and is not recommended for designs in production. Flash should only be left unprotected during debug and development.

By default, PSoC Designer sets each block to protection level 3 - Full Protection. Firmware cannot change the protection levels during runtime; they must be set during the compilation of the hex file and programmed into the device using an external programmer. The protection level of each block can be configured in the PSoC Designer Workspace Explorer by editing the *flashsecurity.txt* file with the appropriate *flashsecurity.txt* character. An example *flashsecurity.txt* file is shown in Figure 5. Each character in the table represents the protection level for one block of flash.

A running program is never prevented from performing internal reads. The `romx` and `index` assembly instructions enable the M8C processor to read from flash for any protection level. Only reads by externally connected testers or programmers may be prevented. In protection modes that prevent external writes, an external programmer can still be used to write a new hex file to the device, but only after an EraseAll operation is completed, which will erase the entire PSoC flash. External write protection will protect against writing individual blocks externally.

It is best to give each flash block the highest protection level allowable for the given application. Typically, Full protection should be used for devices in production, unless the device needs to be able to reprogram its own flash internally. In these cases, protection level 2, Field Upgrade should be used only on the blocks that need to be reprogrammed by the PSoC internally. All other blocks can remain fully protected. Unprotected flash is typically used only during debugging and development.

Figure 5. Example CY8C27xxx flashsecurity.txt File

```
; Edit this file to adjust the Flash security for this project.
; Flash security is provided by marking a 64 byte block with a character
; that corresponds to the type of security for that block, given:
;
; W: Full (Write protected)
; R: Field Upgrade (Read protected)
; U: Unprotected
; F: Factory

; Note #1: Protection characters can be entered in upper or lower case.
; Note #2: Refer to the Flash Program Memory Protection section in the Data Sheet.

; Comments may be added similar to an assembly language comment, by
; Using the semicolon (;) followed by your comment.  The comment extends
; to the end of the line.

; 0 40 80 C0 100 140 180 1C0 200 240 280 2C0 300 340 380 3C0 (+) Base Address

  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 0
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 400
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 800
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address C00
; End 4K parts
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 1000
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 1400
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 1800
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 1C00
; End 8K parts
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 2000
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 2400
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 2800
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 2C00
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 3000
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 3400
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 3800
  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W  W ;    Base Address 3C00
; End 16K parts
```

4 How to Write to Flash

There are two primary predefined methods for writing to flash provided within PSoC Designer:

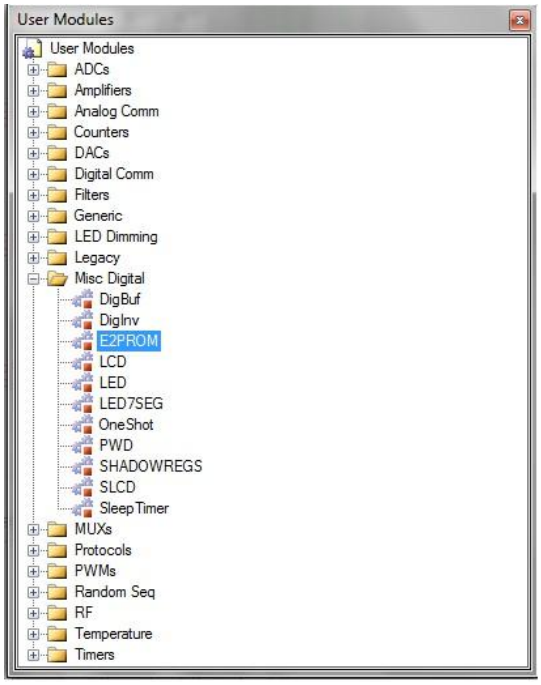
- E2PROM User Module
- The Flashblock API library

Each method, and their pros and cons, are discussed below.

4.1 E2PROM User Module

The E2PROM user module is available in the PSoC Designer User Module Catalog. The user module emulates an E2PROM device within the flash memory of the PSoC. The primary advantage to this user module is that it abstracts the block-oriented flash architecture into something that allows a user to specify 1 to N bytes to be written at a time. Underneath, the E2PROM user module still writes one full block of flash at a time, since this is a requirement of the flash architecture. The user module API handles the process of reading the original content out of a block, changing the byte(s) requested by the user, and writing the new block back to flash. Each write done by the E2PROM user module, even if the write is less than 1 block in length, will consume 1 endurance cycle for the specified block(s) of flash. For more information on using the E2PROM user module and its API's, refer the E2PROM user module datasheet within PSoC Designer.

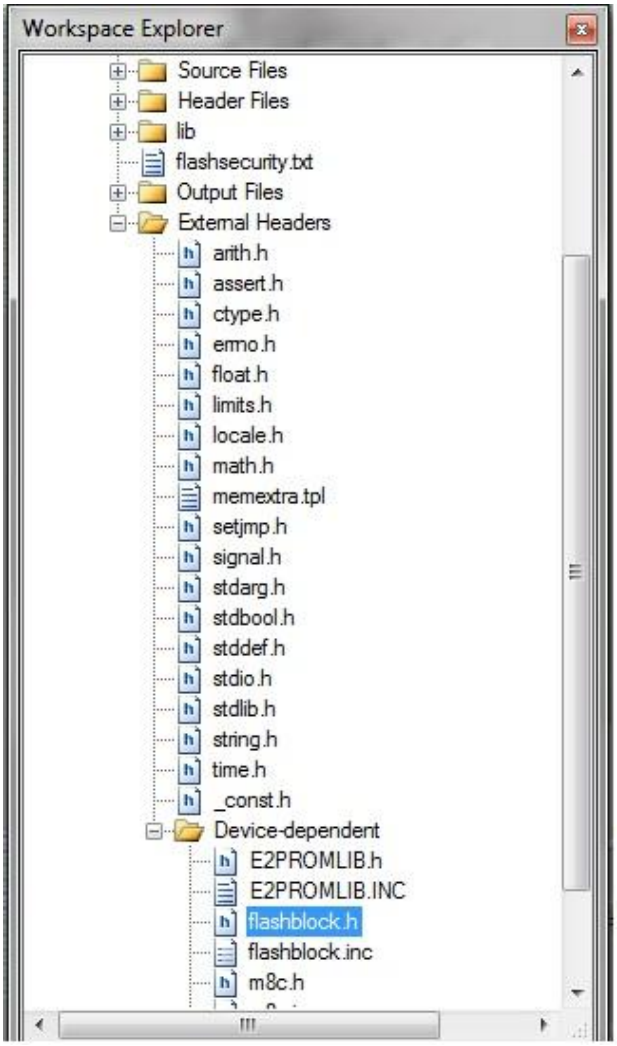
Figure 6. E2PROM User Module



4.2 Flashblock API Library

The Flashblock API library is available for inclusion in any PSoC 1 project simply by including `flashblock.h` (for C projects) or `flashblock.inc` (for assembly projects) in your source file. These header files are found in the External Headers sub-directory in the Workspace Explorer of a PSoC Designer project.

Figure 7. Flashblock API Library



The Flashblock library is a lightweight set of functions that allow a user to easily read and write blocks of flash within a PSoC 1 device. Just like the E2PROM user module, the API's that are provided take care of the low level details (such as calling the SSC functions). However, the Flashblock library will write full blocks at a time, and will not allow partial block writes.

Table 4. Flashblock APIs vs. E2PROM UM

Flash Block API		E2PROM User Module	
Pros	**Cons**	**Pros**	**Cons**
Lightweight (minimal RAM and ROM usage). Only 2 functions, easy to use	Requires full blocks to be updated	Abstracts flash writes to the byte level to allow a user to easily update data in flash less than 1 block long.	Uses more RAM and code space than the Flashblock API library
Flash writes are faster than E2PROM UM (no overhead to manage with partial block writes)		Can be added directly from the user module catalog	Flash writes take longer, with the additional API overhead to manage partial block writes

5 Flashblock API Description

This section describes the contents of the Flashblock API library. The Flashblock library is comprised of two API functions and two structured data types. These elements do not need to be declared by the user; usage of these elements is enabled by including `flashblock.h` (for C projects) or `flashblock.inc` (for assembly projects) in your source file.

Code 1: Read Function Prototype

```
void FlashReadBlock( FLASH_READ_STRUCT *);
```

This function reads a specified flash block to a buffer in RAM. Nothing is returned from this function and a FLASH_READ_STRUCT is passed to the function, which contains the information the function needs to perform the read. The required read structure is shown below.

Code 2: Read Structure (FLASH_READ_STRUCT)

```
typedef struct
{
   // Block Number (0...N) to be read from:
   BYTE   bARG_BlockId;
   // Flash buffer pointer - 2 bytes:
   BYTE * pARG_FlashBuffer;
   // BYTE Read count:
   BYTE   bARG_ReadCount;
}
FLASH_READ_STRUCT;
```

The struct has 3 elements, each of which need to be filled out before passing the struct to the `bFlashReadBlock()` function.

bARG_BlockId: This is the block number to read from. As described in the Flash Architecture section of this document, the blocks of a PSoC 1 device are split into 64 or 128 byte blocks. For devices with 256 or fewer blocks, the value will be 1 byte long and will be called bARG_BlockId. For devices with more than 256 blocks, the value will be 2 bytes long and will be called wARG_BlockId.

pARG_FlashBuffer: 2 byte pointer to the buffer the read data will go into.

bARG_ReadCount: 1 byte value indicating the number of bytes to be read from the block specified in bARG_BlockId. The specified number of bytes will be read starting at address 0 of the bARG_BlockId block. The maximum number of bytes that can be read in a single read is 256. Unlike flash writes, flash reads are not restricted to operating on one full flash block at a time.

Code 3: Write Function Prototype

```
BYTE bFlashWriteBlock(FLASH_WRITE_STRUCT *);
```

This function writes 1 block of data to a specified location in flash. The function returns a byte indicating the result of the flash write. If successful, the returned value will be non-zero. A returned value of 0 indicates a failure occurred. Possible causes for an error are:

1. Protection bits not set properly (full write protection will prevent all flash writes).
2. Voltage below minimum operating voltage of device.
3. Invalid temperature value.

A FLASH_WRITE_STRUCT is passed to `bFlashWriteBlock()`, which contains the information the function needs to perform the write. The required write struct is shown below:

Code 4: Write Structure (FLASH_WRITE_STRUCT)

```
typedef struct
{
   // Block Number (0...N) to be written:
   BYTE   bARG_BlockId;
   // Flash buffer pointer - 2 bytes
   BYTE * pARG_FlashBuffer;
   // Die Temperature, -40 to 100:
   CHAR   cARG_Temperature;
   // Temporary storage (reserved):
   BYTE   bDATA_PWErase;
   // Temporary storage (reserved):
   BYTE   bDATA_PWProgram;
   // Temporary storage (reserved):
   BYTE   bDATA_PWMultiplier;
}
FLASH_WRITE_STRUCT;
```

The struct has 6 elements, 3 of which need to be filled out by the user before passing the struct to the `bFlashWriteBlock()` function:

bARG_BlockId: This is the block number that will be written to. As described in the Flash Architecture section of this document, the blocks of a PSoC 1 device are split into 64 or 128 byte blocks. For devices with 256 or fewer blocks, the value will be 1 byte long and will be called `bARG_BlockId`. For devices with more than 256 blocks, the value will be 2 bytes long and will be called `wARG_BlockId`.

pARG_FlashBuffer: 2 byte pointer to the buffer holding the data to be written to flash. Regardless of the size of the buffer, the write routine will always write one full block of data (either 64 or 128 bytes, depending on the device). If the buffer is less than 1 block in length, then the rest of the block will be written with whatever data follows the buffer in RAM.

cARG_Temperature: *One* byte value indicating the temperature, in degrees Celsius, of the PSoC dies during the flash write. This value should be within the operational temperature range (available in the device datasheet) of the device in use, or else a flash write failure could occur. See the Selecting the Appropriate Temperature section of this document for additional information on setting this value.

bData_PWErase: Temporary storage variable; should not be set by the user.

bDATA_PWProgram: Temporary storage variable; should not be set by the user.

bDATA_PWMultiplier: Temporary storage variable; should not be set by the user.

6 Example Projects

This Application Note has 3 example projects for the user to jump start on the implementation.

HW Requirements: CY3210- PSoC Eval1

SW Requirements: PSoC Designer 5.2 SP1 or later

PSoC 1 Device requirements: CY8C29466-24PXI, CY8C27443-24PXI

User Modules: E2PROM, LCD

API Library: Flashblock

Functions:

- Write (“AN2015 E2PROM RW” or “AN2015 FLASH RW”) to the last block of the flash based on the methodology used in that example project.
- Read back the written values and display it on the LCD

- On button press, scramble the buffer (“RW E2PROM AN2015” or “RW FLASH AN2015”) based on the methodology used in that example project and execute another write to the same block of flash.
- Read back the written values and display it on the LCD

Table 5. Example Projects Differences

Example Project	Device	Methodology	Flash (B)	RAM (B)	Comments
1	CY8C29466-24PXI (512 blocks of flash)	E2PROM User Module	2263	37	Showcases E2PROM UM usage
2	CY8C29466-24PXI (512 blocks of flash)	Flashblock API Library	1754	45	Showcases E2PROM UM usage for 512 block flash devices
3	CY8C27443-24PXI (256 blocks of flash)	Flashblock API Library	1451	43	Showcases Flashblock API usage for 256 block flash devices

6.1 Hardware Setup / Demo

- Connect a wire between SW and P0[2] in CY310-PSoC Eval1
- Place the right part number as per the example project
- The image of the HW setup is shown in Figure 8,
- Connect Miniprog 1 on Jumper J11 and program the target PSoC 1 device with the required hex file.
- Once programming is done, power the board from PSoC Programmer
- On power-up, a character array is written to the flash using E2PROM method and read back and displayed on the LCD.
- If the switch ‘SW’ is pressed now, then the character array is scrambled and written again. Then it is read back and displayed on the LCD.
- The snapshot is provided for the first example project where E2PROM is used. The demo procedure is same for all the example projects.

Figure 8. Hardware Setup

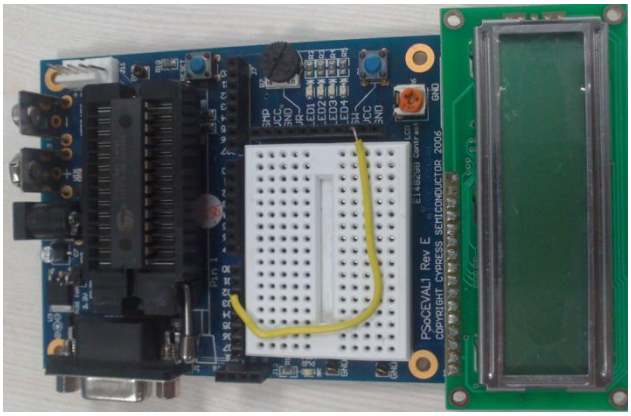


Figure 9. Screenshot of the LCD Message after First Write

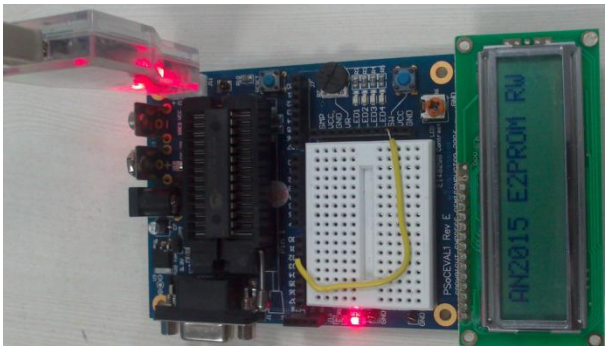
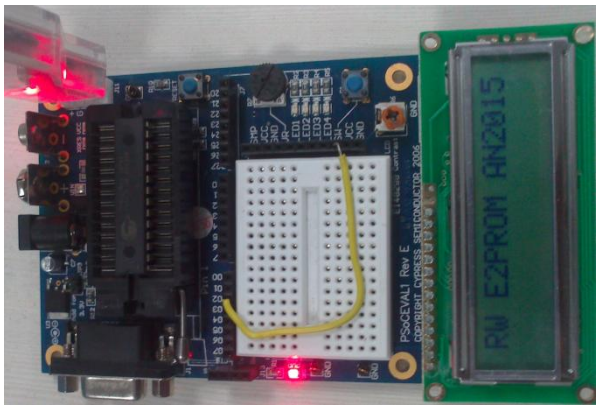


Figure 10. Snapshot of the LCD Screen after the Scrambled Second Write



6.2 Example Project 1: E2PROM – CY8C29466-24PXI

E2PROM User module is used in this project with the following parameters.

- FirstBlock is chosen as Block # 511 in this example project
- Length is chosen as 16 bytes for this example project.

Figure 11. E2PROM UM Parameters

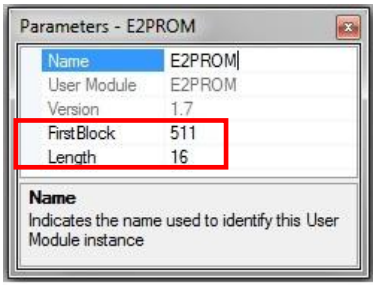


Figure 12. Variable Initialization

```
/*****************************************************************************
*  Include Files
*****************************************************************************/
/* Part specific constants and macros */
#include <m8c.h>

/* PSoC API definitions for all User Modules */
#include "PSoCAPI.h"

/* Prototypes of all string functions */
#include <stdlib.h>

/* E2PROM definitions and macros */
#include <E2PROM.h>

/*****************************************************************************
*  Defines
*****************************************************************************/
/* Room Temperature for the E2PROM writes */
#define Temp 25

/* checks if SW is pressed (pulled-down by default) */
#define SW_Press PRT0DR & 0x04

/*****************************************************************************
*  Global Varibles
*****************************************************************************/
/* Write buffer to store the data to be written to E2PROM */
unsigned char Write_Buffer[16];

/* Read Buffer to store the data read from E2PROM */
unsigned char Read_Buffer[16];
*/
```

Figure 13. E2PROM Write and Read

```
/* Write the buffer(WriteData) data to E2PROM assuming room temp to be 25C */
bError = E2PROM_bE2Write(0,(BYTE *)&Write_Buffer, sizeof(Write_Buffer), Temp);

/* If the write operaiton is successful, read back the data to the ReadBuffer */
if ( bError == E2PROM_NOERROR )
{
    /* Read the data back from E2PROM using the E2Read API function */
    E2PROM_E2Read( 0, (BYTE *)&Read_Buffer, sizeof(Read_Buffer) );
    /* Display the read data on the LCD */
    LCD_Position(0,0);
    LCD_PrString(Read_Buffer);
}
```

6.3 Example Project 2: Flashblock – CY8C29466-24PXI

Figure 14. Flashblock API Variables

```
/*****************************************************************************
*  Defines
*****************************************************************************/
/* Temperature for flash write is set here */
#define Temp 25

/* SW is pulled down by default connected to P0[2] */
#define SW_Press PRT0DR & 0x04

/* Block ID used for flash writes and read  */
#define BLOCK 511

/* Return value when an error occured during the flash block write function */
#define FLASH_ERROR 0

/* size of the array to be written or read */
#define SIZE 16

union
    {
        FLASH_WRITE_STRUCT Write;
        FLASH_READ_STRUCT Read;
    } FlashParams;

/*****************************************************************************
/*  Global Varibles
/*****************************************************************************/
/* Buffer to store the values to be written to the flash */
unsigned char Write_Buffer[16];

/* buffer to store the values to be read from the flash */
unsigned char Read_Buffer[16];
```

Figure 15. Flashblock Read and Write Function for 512 Block Flash Device

```
FlashParams.Write.wARG_BlockId = BLOCK;             /* Block ID  */
FlashParams.Write.pARG_FlashBuffer = Write_Buffer;  /* Start with the first byte of Buffer */
FlashParams.Write.cARG_Temperature = Temp;          /* Place your average expected device temperature here.
                                                       For optimal flash write conditions the temperature needs to be
                                                       within +/- 20 C of the actual device die temperature.*/
/* bFlashWriteBlock returns a non-zero value in the case of a flash write failure.
   See flashblock.h line 55 for more details.*/
/* Write the buffer(WriteData) data to Flash assuming room temp to be 25C  */
bError = bFlashWriteBlock(&FlashParams.Write);
/* If the write operaiton is successful, read back the data to the ReadBuffer */
if ( bError != FLASH_ERROR )
{
    /* Enable global interupts */
    M8C_EnableGInt;

    /* Starting at the first byte of the first block */
    FlashParams.Read.wARG_BlockId = BLOCK;

    /* RAM buffer to read */
    FlashParams.Read.pARG_FlashBuffer = Read_Buffer;

    /* 16 byte read */
    FlashParams.Read.wARG_ReadCount = SIZE;

    /* read the data from the flash to the buffer */
    FlashReadBlock(&FlashParams.Read);

    /* Display the read data on the LCD */
    /* Fix the position on the LCD */
    LCD_Position(0,0);

    /* Print the characters in the LCD */
    LCD_PrString(Read_Buffer);
}
```

- For Flash Write, BlockId, pointer to the flash write buffer and Temperature are set and then `bFlashWriteBlock()` function is called.
- For Flash Read, BlockId, pointer to the flash read buffer and ReadCount are set and then `bFlashReadBlock()` function is called.
- BlockId parameter for Flash Write and Readcount parameter for Flash Read are declared as Word as the number of flash blocks is 512 and cannot be accommodated in a BYTE variable.

6.4 Example Project 3: Flashblock – CY8C27443-24PXI

Figure 16. Flash Read and Write for 256 Block Flash Device

```
FlashParams.Write.wARG_BlockId = BLOCK;             /* Block ID  */
FlashParams.Write.pARG_FlashBuffer = Write_Buffer;  /* Start with the first byte of Buffer */
FlashParams.Write.cARG_Temperature = Temp;          /* Place your average expected device temperature here.
                                                       For optimal flash write conditions the temperature needs to be
                                                       within +/- 20 C of the actual device die temperature.*/
/* bFlashWriteBlock returns a non-zero value in the case of a flash write failure.
   See flashblock.h line 55 for more details.*/
/* Write the buffer(WriteData) data to Flash assuming room temp to be 25C  */
bError = bFlashWriteBlock(&FlashParams.Write);
/* If the write operaiton is successful, read back the data to the ReadBuffer */
if ( bError != FLASH_ERROR )
{
    /* Enable global interupts */
    M8C_EnableGInt;

    /* Starting at the first byte of the first block */
    FlashParams.Read.wARG_BlockId = BLOCK;

    /* RAM buffer to read */
    FlashParams.Read.pARG_FlashBuffer = Read_Buffer;

    /* 16 byte read */
    FlashParams.Read.wARG_ReadCount = SIZE;

    /* read the data from the flash to the buffer */
    FlashReadBlock(&FlashParams.Read);

    /* Display the read data on the LCD */
    /* Fix the position on the LCD */
    LCD_Position(0,0);

    /* Print the characters in the LCD */
    LCD_PrString(Read_Buffer);
}
```

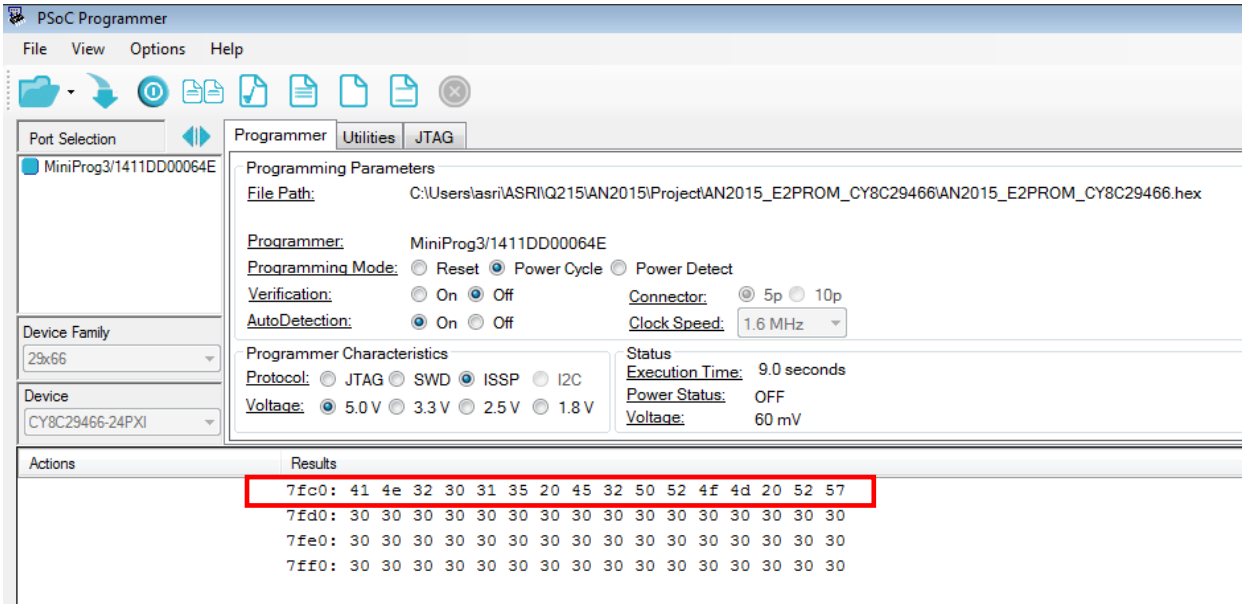
- BlockId parameter for Flash Write and Readcount parameter for Flash Read are declared as BYTE as the number of flash blocks is 256.
- CY8C28 family is an exception, where these variables are declared as WORD even though the device has only 256 blocks of flash.

6.5 Reading Data from Flash Using PSoC Programmer

To verify the flash contents for the example projects explained above, PSoC Programmer can be used.

- Launch PSoC Programmer 3.14 or later
- Connect Miniprog 1 between the PC and CY3210-Jumper J11.
- Make sure one of the three example projects are loaded into the right PSoC 1 device on board.
- Now, press F7 to read the flash contents

Figure 17. Flash Read Using PSoC Programmer for Example Project 1 After First Write



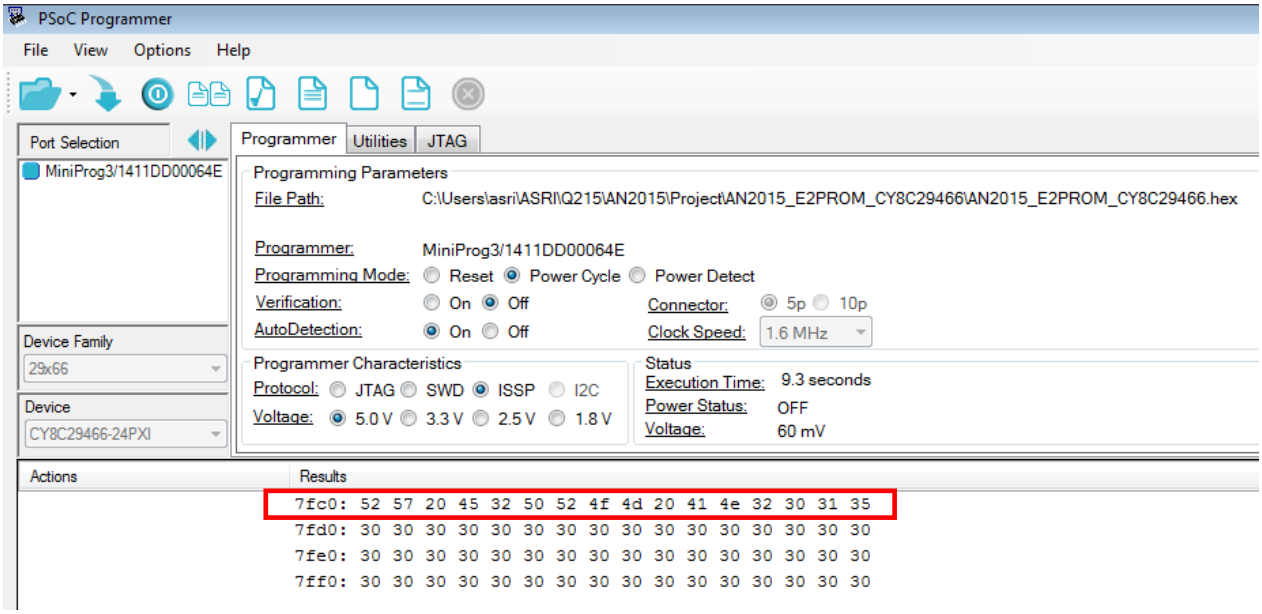
- Each flash block is 64 bytes in PSoC 1 device.
- There are a total of 512 blocks in CY8C29466-24PXI, which have 32KB of flash.
- E2PROM was configured for the last block ‘511’ and length as 16 bytes.
- 511th block address in hex is ‘0x7FC0’, which is seen above in the snapshot.

Table 6. ASCII Table for Characters

Char	ASCII in hex	Char	ASCII in hex	Char	ASCII in hex
A	41	E	45	F	46
N	4e	2	32	L	4c
2	32	P	50	A	41
0	30	R	52	S	53
1	31	O	4f	H	48
5	35	M	4d	R	52
Space	20	Space	20	W	57

- You can perform the read after the first write and second write and verify if the flash contents have changed accordingly.
- Figure 18 has the snapshot after the second write to the flash.
- In flashsecurity.txt file in the example project, only the last block’s protection mode is modified as ‘U’ which means it is unprotected. So, it can be read and written from external as well as internal. The rest of all the blocks are ‘W’, which means fully protected. So, it cannot be read and, hence, is ‘xx’ in the previous blocks.

Figure 18. Flash Read Using PSoC Programmer for Example Project 1 After Second Write



7 Special Considerations

This section describes a few additional considerations that often come up when designing a project that incorporates flash writes. Things to consider include interrupts, voltage stability and placing data in flash. Additional information on many of the topics discussed in this section can be found in the Technical Reference Manual and device datasheet.

7.1 Interrupts and Timing

Calling `bFlashWriteBlock()` globally disables interrupts while erasing and writing the specified block. This is done within the hard-coded SROM code within each PSoC device. This ensures the write process is not interrupted, guaranteeing proper timing of the erase/write pulse widths and preventing any accesses to partially written flash. Also, during the execution of the actual flash write, the CPU clock will be changed to a known rate (either 12 MHz or 6 MHz, depending on the SLIMO setting). This allows the flash write function to generate proper pulse width timing, no matter what the user CPU speed is.

For additional information on the nominal erase and write timing for a particular PSoC device, see the AC Programming Specifications section of the device datasheet. Note the timing specified in the datasheet is the typical timing of the pulse widths used to erase and write a block of flash and do not include the overhead required to execute the full flash write function.

7.2 Voltage Stability

The supply voltage (V_{DD}) must be within the valid operating region during a `bFlashWriteBlock()` operation. It is best to properly use the power-on reset (POR) circuit so that a write operation does not occur if V_{DD} decreases below minimum operating voltage. The code located in boot.asm (an automatically generated file) properly sets the correct POR level based on the CPU's operating frequency. It is best not to change the POR level in user code. If the voltage supply is not properly maintained during a write operation, a reset may occur, and the data within the block may not be written correctly and there is no indication of the write failure.

PSoC offers a low-voltage detect (LVD) interrupt that will allow an interrupt to occur if the voltage ever falls below the POR level. Since interrupts are disabled during the actual flash write, this interrupt will not detect any power issues while the write is in progress, but can still be useful in detecting any low voltage issues before or after a flash write occurs.

7.3 Placing Data in Flash

Placing data at a specific location in flash serves several useful purposes when reading from and writing to flash:

- Allows predefined values to be loaded into the flash during compilation and programming, if needed.
- Reserves the flash sections to prevent the compiler from being able to place user code or other data in the same portion of flash memory.
- Allows flash data to be read by accessing declared names rather than using the flash read function.

Code 5 shows an example of how to place a piece of data in flash memory at a specific address using C. This allows the data to be read by referencing the variable name and allows the data to be updated by using the flash write API to write the block of data where the data is stored. In this example address 0x3FC0, or block 255, is used:

Code 5: C Data at Fixed Location in Flash

```
#pragma abs_address: 0x3FC0
const BYTE myArray[64] = {0};
#pragma end_abs_address
```

Once this declaration is made, this data can easily be accessed by referencing `myArray[x]`. When the data needs to be updated, the `bFlashWriteBlock()` routine can be used to update block 255. This technique is also useful because it prevents any code or other user data from being placed in the region of flash, preventing data corruption.

The same functionality can be achieved using assembly, but requires a little bit more work. First, each declared piece of data must be placed in its own linker area. In the example shown in Code 6, data is added to a new linker area called ‘myArea’. Any custom area name can be used, but the flags (REL, CON, ROM) shown in the example should remain the same to ensure the area is added to flash.

Code 6: Adding Data to an AREA in Flash (Assembly)

```
;Declare new area in ROM
AREA myArea (REL,CON,ROM)
;Add data to AREA
myArray:: blk 64
;Switch back to default area (text)
AREA text
```

Once the desired data has been declared in a user defined area, the active area should be switched back to the default (text) area. This ensures the rest of the code in the project goes into the correct memory areas.

Lastly, when the data has been declared in the new user defined area, a custom linker option must be used to place the new area at a specific location in flash. This is done by adding a file called *custom.lkp* to the main project folder (the folder containing main and any other user created source files). Within *custom.lkp*, the following command should be used:

Syntax:

```
-b<area name>:<start address>.<end address>
```

Example:

```
-bmyArea:0x3FC0.0x3FFF
```

In this example, the user defined area called ‘myArea’ is placed at address 0x3FC0, or block number 255, in flash. Similar to the C example, this data can easily be accessed by referencing myArray instead of using the flash read function. When the data within myArray needs to be updated, the `bFlashWriteBlock()` routine can be used to update block 255.

For additional information on the #pragma and other linker directives, refer the C Language Compiler User Guide available in the PSoC Designer Documentation folder.

8 Summary

This application note describes the basic flash architecture present in PSoC 1 and covers the various techniques available for reading from and writing to flash. Using the information presented in this application note, users will be able to store nonvolatile data in flash with minimal effort.

9 References

- Section 3.12 through 3.14 of PSoC Programmer User Guide documents the procedure to read the flash contents, verify checksum and erase all or selective blocks of Flash.
- PSoC® 1 ISSP Programming Specifications - CY8C21x23, CY8C21x34, CY8C23x33, CY8C24x23A, CY8C27x43, CY8CTMG110, CY8CTST110
- PSoC® 1 ISSP Programming Specifications - CY8C21x45, CY8C22x45, CY8C24x94, CY8C28xxx, CY8C29x66, CY8CTST120, CY8CTMA120, CY8CTMG120, CY7C64215

About the Author

Name: Praveen Kumar M

Title: Applications Engineer Sr

Document History

Document Title: AN2015 – PSoC® 1 – Getting Started with Flash & E2PROM

Document Number: 001-40459

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1532004	VED	10/02/2007	New publication of existing application note.
*A	2625181	BTK	12/19/2008	Modified title. Updated content to include devices with more than 16 kB of flash. Modified content to point to flash endurance specifications in data sheets. Corrected and added more information regarding flash protection information. Updated content with minor formatting and grammatical improvements.
*B	2978865	RLRM	07/12/2010	Updated the example code in the Appendix.
*C	3190554	DRSW	03/08/2011	Updated title to read “AN2015 - PSoC® 1 - Reading and Writing Flash”. Updated Software Version in page 1 to PSoC® Designer™ 5.1.
*D	3308489	DRSW	07/11/2011	Updated flash write time information in the Flash Write Algorithm section. Updated Table 1 to include the characters used in flashsecurity.txt Noted that some PSoC devices will have a 1 byte block ID parameter instead of a 2 byte word.
*E	3405282	DRSW	10/13/2011	Full rewrite. Added information on E2PROM UM Many sections given their own heading and expanded (architecture, temperature, flash protection, algorithm, Flashblock API examples, and special considerations). Updated template.
*F	3421756	DRSW	10/25/2011	Minor changes - page header fixed
*G	3673054	PRKU	07/23/2012	Updated title to read as “PSoC® 1 – Getting started with Flash & E2PROM”. Updated Software Version to “PSoC® Designer™ 5.2 SP1”. Updated Abstract. Updated Introduction (Updated Flash Architecture (Added Table 1), updated Flash Write Algorithm (Updated description), updated Selecting the Appropriate Temperature (Updated description, added Table 2), updated Flash Protection (Updated description)). Updated How to Write to Flash (Updated E2PROM User Module (Updated description, added Figure 6), updated Flashblock API Library (Updated description, added Figure 7), added Table 4). Removed the section “Flashblock API Usage”. Added Example Projects. Tested both the example code with PD 5.2 Added References Updated in new template. Completing sunset review.
*H	4764479	ASRI	05/13/2015	Updated template
*I	5687791	AESATMP8	04/19/2017	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