E2PROM Datasheet E2PROM V 1.7

Copyright © 2004-2013 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	flash	RAM	
CY8C29x66, CY8CLED16, CY8CPLC20, CY8CLED16P01	0	0	0	26+1013 +Size	See Below	0
CY8C27x43, CY8C24x94, CY8C22x13, CY7C64215, CY8CLED04/08, CY8CLED0xD, CY8CLED0xG, CY8C22x45, CY8C28x45, CY8C28xxx	0	0	0	25+750 +Size	See Below	0
CY8C24x23A, CY8C23x33, CY8C21x23, CY8CLED02	0	0	0	25+750 +Size	See Below	0
CY8C21x34, CY7C603xx, CYWUSB6953, CY8C20x24, CY8C20x34, CY8C21x45, CY8C21x12	0	0	0	25+1044 +Size	See Below	0

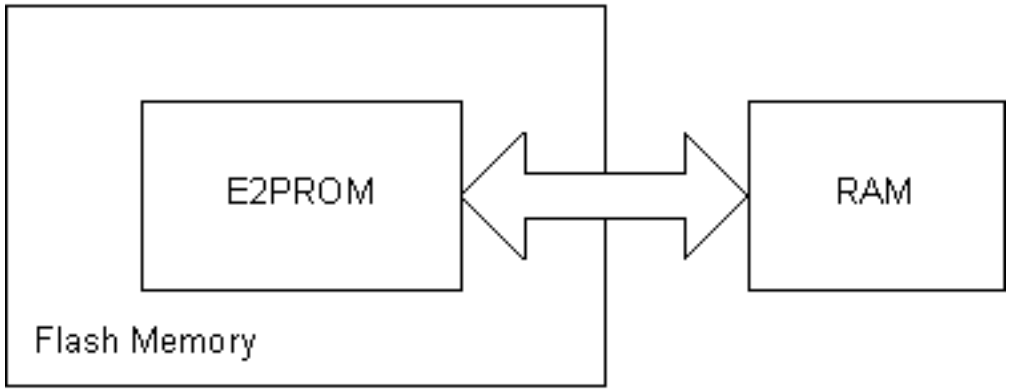
For one or more fully configured, functional example projects that use this user module go to www.cypress.com/psocexampleprojects.

Features and Overview

- Full byte-oriented EEPROM emulation
- Abstracts block-oriented flash architecture
- Efficient use of memory

The EEPROM User Module emulates an EEPROM device within the flash memory of the PSoC device. The EEPROM device can be defined to start at any flash block boundary, with a byte length from 1 to the remainder of flash memory space. The API enables the user to read and write 1 to N bytes at a time.

Figure 1. EEPROM Block Diagram



Functional Description

The EEPROM User Module is a software algorithm that uses no hardware resources of the PSoC device. One or more instances of these EEPROM virtual devices can be created.

The flash is organized per device as 511 blocks of 64 bytes for a 32K device, 256 blocks of 64 bytes for a 16K device, 128 blocks of 64 bytes for an 8K device, and 64 blocks of 64 bytes for a 4K device. The architecture of the PSoC allows the flash data to be read on a byte-by-byte basis, but requires the data to be written on a block-by-block basis – 64 bytes at a time. The intent of this user module is to emulate an EEPROM device (a byte-read, byte-write oriented device) on a flash-based memory device (a byte-read, block-write oriented device).

The EEPROM storage area starts on a flash memory block boundary and consists of 1 or more bytes. This virtual device is accessed by using the E2Read() and E2Write() API routines. The virtual address space is from 0 to N-1, where N is the length/size in bytes of the EEPROM device. The size of the area reserved for storage must also be accounted for in flash memory usage but this quantity cannot be calculated by the resource meter. This Size element is identified as ‘Size’ in the memory usage table above.

The E2Read() API algorithm efficiently reads the flash memory byte-by-byte using the ROMX M8C instruction. This algorithm requires use of the last 8 bytes of RAM: 0xF8 to 0xFF.

The E2Write() API algorithm writes data to the flash memory on a block-by-block basis. Based on the starting address offset into the EEPROM memory space, the E2Write() routine parses the data to be written into segments that are aligned on block boundaries. This algorithm also requires use of the last 8 bytes of RAM: 0xF8 to 0xFF.

For segments that span an entire block, 64 bytes, the FlashBlockWrite() routine is called and the block is written.

For segments that are less than the 64 bytes, a temporary 64-byte stack buffer is composed of both the unwritten and modified data. After the data is written, the buffer is released from the stack. This is required to preserve the data within the block.

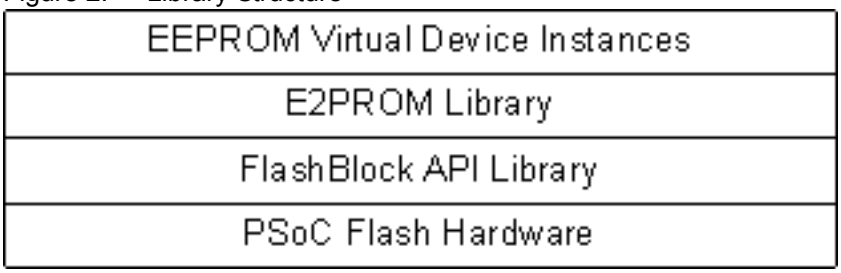
To reduce system overhead by optimizing parameters, see the *Parameters and Resources* section ahead.

Software Organization

The following figure shows how the software is organized into several layers consisting of two libraries and N EEPROM virtual device instances.

Resource usage above lists the API library size in flash plus the additional code contained in the FlashBlock API Library and the E2PROM library. In cases where multiple instances of this device are used only the UserModule E2PROM API length is calculated in the resource meter. Only one FlashBlock API Library and E2PROM Library are used for any number of user module instances.

Figure 2. Library Structure



The FlashBlock API Library provides APIs that enable basic flash block read and write routines. These routines interface directly with the hardware and cause the device to enter System Supervisor mode. All interrupts are masked during this period. This library is only linked once, for all instances of the EEPROM virtual devices instantiated.

The E2PROM Library translates block operations into byte-wise operations by way of block alignment and buffering. It then calls the FlashBlock API Library functions to write to the flash. This library is only linked once, regardless of the number of EEPROM virtual devices.

The EEPROM virtual device layer provides a simple means to allow multiple instances of an EEPROM device, while only incurring the overhead of a single instance of both the E2PROM Library and the FlashBlock API Library. Each instance of the EEPROM virtual device contains a customized copy of this source code, which is minimal at only 16 bytes.

Both the E2PROM and the FlashBlock API libraries are included in a library file. This file automatically links into the project if calls are made to these functions.

Note These libraries are not re-entrant.

Flash Write Cycle Longevity

The flash memory has a limit with respect to the total number of lifetime writes. This device-specific total number can be found in the applicable device family datasheet.

Flash Protection

The *flashsecurity.txt* file must be edited to allow flash writes to the blocks that comprise the EEPROM device.

There is one ASCII character in flashsecurity.txt for each flash block of the device. The valid options for this ASCII character are 'W', 'R', 'U', and 'F'. The ASCII characters in this file define the flash security settings that are applied to each flash block. A 'W' fully protects the flash block from any writes and is the default setting. To allow the E2PROM User Module to read and write blocks the protection option should be changed to 'R', 'F', or 'U'. The option should be changed for each flash block that is occupied by the E2PROM. To properly change the security settings in *flashsecurity.txt*, delete the character corrsponding to the flash block to be affected. Type in a new character corresponding with the desired security setting. Save the project and rebuild it to ensure that the changed settings have taken effect.

Mode	Settings	Description	In PSoC Designer
00b	SR ER EW IW	Unprotected	U = Unprotected
01b	SR ER EW IW	Read protect	F = Factory upgrade
10b	SR ER EW IW	Disable external write	R = Field upgrade
11b	SR ER EW IW	Disable internal write	W = Full protection

Efficient Memory Use

The following guidelines can be used to maximize the efficiency of both RAM and flash resources.

If Flash Memory is Plentiful (Efficient use of RAM)

- Create EEPROM devices in multiples of 64-byte lengths, even if the data to be stored is less.
- Always write the data in multiples of 64-byte lengths.
- Always read the data with the correct byte count.

For Example, Serial Number Data is 10 bytes in length. Create an EEPROM User Module with the name serial number. Set the Length parameter to 64. This allocates a 64-byte block in flash memory. In RAM, set up a 10-byte data array to hold the serial number data. When writing the serial number via E2Write(), specify the wByteCount parameter as 64. However, when reading the serial number data, specify the wByteCount parameter as 10.

If Flash Memory is at a Premium (Efficient use of Flash)

- Create EEPROM devices using only the required number of bytes.
- Always write the data with the correct byte count.
- Always read the data with the correct byte count.

Timing

The timing of the EEPROM algorithms is summarized in the following table:

Table 1. Unless otherwise specified in the following table, all limits guaranteed for T_A = 25°C, V_{dd} = 5.0V.

Parameter	Conditions and Notes	Typical	Limit	Units
Time to Write 0°C-100°C	Time to write depends on the number of flash blocks that require an update.[1,2]	45	120[3,4]	ms/Block
Time to Write -40°C-0°C	Time to write depends on the number of flash blocks that require an update.[1,2]	70	240[3,4]	ms/Block
Time to Read	81 + 46N Clock Cycles	--	--	Clks/Bytes
	Worst Case is 1 Byte:	--	127	
	For 16 Bytes: (81+46(16))/10	51	--	
	For 64 Bytes: (81+46(64))/64	48	--	

Timing Table Notes

1. Interrupts are masked during flash read and write operations during the time the M8C is in System Supervisor mode.
2. Time to Write is determined by the FlashBlockWrite() routine to erase and write each flash block.
3. Total number of flash writes is limited. See the part-specific datasheet for specifications on number of write cycles per block.
4. Limit is specified for normal programmed operation. In emulation mode using the debugger, the write pulse may take up to 1 to 2 seconds.

DC and AC Electrical Characteristics

See the device datasheet for your PSoC device for flash characteristics and specifications.

Placement

The EEPROM User Module is implemented in software and does not require placement.

Parameters and Resources

FirstBlock

The FirstBlock is the starting block in which the EEPROM device resides. Values range from 0 to the maximum number of flash blocks for the specific device: 511 for 32K devices, 255 for 16K devices, 127 for 8K devices, 63 for 4K devices, and 31 for 2K devices.

Special care must be taken to ensure that the E2PROM User Module's flash memory location does not overlap any code or other important data located in the flash. You should locate the E2PROM memory location as high as possible in the flash memory. For instance, if the device has 512 blocks of flash, locating the E2PROM User Module at flash block 511 would be a good choice. Locating it at flash block 0 or 1 would likely cause a system failure, since the E2PROM location overlaps the interrupt vectors. Any write operation to an E2PROM at block 0 or 1 would overwrite the interrupt vector table. Locating the E2PROM near the middle of the flash memory is also not a good practice. The ImageCraft compiler does not reserve a separate memory area for the E2PROM. Therefore, as your code gets larger, it eventually overlaps the E2PROM location near the middle of the flash memory.

Length

The length defines the size, in bytes, of the EEPROM device. The valid range is from 1 to N, where N is the size of EEPROM virtual device in bytes.

Warning: Ensure that the physical location of the virtual EEPROM device does not exceed the flash size of the device. The first physical byte of the EEPROM device is located at FirstBlock * 64. The last byte of the virtual device is physically located at FirstBlock * 64 + Length - 1. Error checking is not performed. Failure to ensure that the size is not exceeded may result in unpredictable flash writes to unintended blocks.

RAM Memory Overhead

The following RAM memory resources are utilized:

API Routine	Memory Utilization
E2Read()	Stack Space: 8 Bytes RAM High Memory: 0xF8 – 0xFF
E2Write() with NO Partial Block Writes	Stack Space: 32 Bytes RAM High Memory: 9 Bytes 0xF8 – 0xFF
E2Write() with Partial Block Writes	Stack Space: 103 Bytes RAM High Memory: 9 Bytes 0xF8 – 0xFF

Common Problems

1. Watch Dog Timer (WDT) Usage

   When the write operation to E2PROM is in progress, a device reset may improperly terminate the write operation. This can lead to data corruption or loss of data in the E2PROM block. As a result, if WDT is enabled in the application, it must be cleared before the E2PROM write is initiated in the application firmware, and you must select a WDT time out period that is more than the worst case E2PROM write time mentioned in the user module datasheet. This ensures that device does not reset from WDT time out during the E2PROM write operation.

2. Improper Settings in Flashsecurity.txt

   If any flash block is used as an E2PROM block in a project, the flashsecurity.txt file in the project must be modified to remove the write protection for respective flash blocks. If the protection status of the flash block is the not changed to "unprotected", write to E2PROM blocks will not be successful.

3. Power Stability

   Stable VDD must be ensured during the E2PROM erase write process. Each device has a different supported VDD for flash operation. As a result, the device datasheet must be referred to for the supported VDD range.

Application Programming Interface

The Application Programming Interface (API) routines are provided as part of the user module to allow the designer to deal with the module at a higher level. This section specifies the interface to each function together with related constants provided by the “include” files.

Note

In this, as in all user module APIs, the values of the A and X registers may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This “registers are volatile” policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy, too. Though some user module API functions may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

The EEPROM User Module APIs are described as follows.

E2PROM_Start

Description:

A null function, maintained for user module API consistency.

C Prototype:

```
void  E2PROM_Start(void)
```

Assembler:

```
lcall  E2PROM_Start
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model in the CY8C29xxx. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

E2PROM_Stop

Description:

A null function, maintained for user module API consistency.

C Prototype:

```
void  E2PROM_Stop(void)
```

Assembler:

```
lcall  E2PROM_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model in the CY8C29xxx. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

E2PROM_bE2Write

Description:

Writes the specified data to the defined EEPROM from a RAM buffer. Note that the *flashsecurity.txt* file must be set properly to allow writing to flash.

C Prototype:

```
CHAR  E2PROM_bE2Write(WORD wAddr, BYTE *pbData, WORD wByteCount,
                      CHAR cTemperature);
```

Assembly:

```
push  X
mov   X, SP
mov   A, <cTemperature>               ; cTemperature argument
push  A
mov   A, <wByteCount>                 ; wByteCount – MSB
push  A
mov   A, <wByteCount+1>               ; wByteCount - LSB
push  A
mov   A, <pbData>                     ; pbData – MSB
push  A
mov   A, <pbData+1>                   ; pbData – LSB
push  A
mov   A, <wAddr>                      ; wAddr - MSB
push  A
mov   A, <wAddr+1>                    ; wAddr - LSB
push  A
lcall  E2PROM_bE2Write
add   SP, -E2_WR_ARG_STACK_FRAME_SIZE ; restore call stack
pop   X
```

Where <..> refers to any addressing mode or number of instructions to place referenced data into the Accumulator.

Parameters:

wAddr: Address offset of the EEPROM device address space from which the RAM data is written. It can be 0 to the N-1, where N is the length of the EEPROM device.

pbData: Pointer to the RAM buffer that contains data to write.

wByteCounter: Number of bytes to write to flash.

cTemperature: Temperature of the PSoC die, in degrees Celsius. This value can be specified by using one of the following:

- A user module, such as FlashTemp.
- An external device or sensor.
- A nominal value that applies to all environmental conditions that the PSoC device experiences.

  For example, room temperature = 25°C.

Return Values:

The following values can be returned:

Return Flag	Description	Value
NOERROR	Successful completion of operation.	0
FAILURE	Unsuccessful completion of operation. Most likely a result of flash-protection bit errors.	-1
STACKOVERFLOW	Stack space was not sufficient for algorithm requirements.	-2

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model in the CY8C29xxx. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP, IDX_PP and MVW_PP page pointer registers are modified.

E2PROM_E2Read

Description:

Reads the specified EEPROM device data from flash into the specified RAM buffer.

C Prototype:

```
void E2PROM_E2Read(WORD wAddr, BYTE *pbData, WORD wByteCount)
```

Assembler:

```
push  X
mov   X, SP
mov   A, <wByteCount>                 ; wByteCount – MSB
push  A
mov   A, <wByteCount+1>               ; wByteCount - LSB
push  A
mov   A, <pbDataDest>                 ; pbDataDest – MSB
push  A
mov   A, <pbDataDest+1>               ; pbDataDest – LSB
push  A
mov   A, <wAddr>                      ; wAddr - MSB
push  A
mov   A, <wAddr+1>                    ; wAddr - LSB
push  A
lcall   E2PROM_E2Read
add   SP, -E2_RD_ARG_STACK_FRAME_SIZE  ; restore call stack
pop   X
```

Where <..> refers to any addressing mode or number of instructions to place referenced data into the Accumulator.

Parameters:

wAddr: Address offset of the EEPROM device address space from which the flash data is read. It can be 0 to the N-1, where N is the length of the EEPROM device.

pbData: Pointer to the RAM buffer that data is read into.

wByteCounter: Number of bytes to read from flash.

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model in the CY8C29xxx. When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP and MVW_PP page pointer registers are modified.

Sample Firmware Source Code

The following code is C language source code:

```
//*****************************************************************************
//*    EEPROM User Module Example Code:
//*
//*    A SerialNumber EEPROM was created to start at block 250 with a length
//*    of 10 bytes. Remember to edit the Flashsecurity.txt file to allow
//*    writes to flash  (Set the 250th block to U - Unprotected).
//*
//*    This example:
//*
//*        a) Writes the initial data to the EEPROM block area
//* Note that this will invoke the SavePartial algorithm which
//* allocates a temporary 64-byte buffer on the stack. If
//* Flash memory is plentiful and the extra 54 bytes can be
//* wasted, set the SerialNumber device to a length of 64 and when
//* writing, specify a bytecount of 64. This will write an entire
//* block without using a temporary buffer. The extra 54 bytes
//* beyond the SerialNumber will be bogus data.
//*
//*        b) Reads the data back into a RAM buffer
//*
//*****************************************************************************/

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

#define  ADDRESS_OFFSET    0
#define  NUMBER_OF_BYTES   10
#define  TEMPERATURE       25

/* Initialize a RAM buffer with default Serial Number */
BYTE  abInitialSerialNumber[] = {'0','1','2','3','4','5','6','7','8','9' };
BYTE  abSerialNumberBuffer[NUMBER_OF_BYTES];

void main(void)
{
    BYTE bError;


 /* Write the Serial Number - assume temp of 25C */
    bError = E2PROM_bE2Write(ADDRESS_OFFSET, abInitialSerialNumber, NUMBER_OF_BYTES,
TEMPERATURE);


 if ( bError == E2PROM_NOERROR )
    {
        /* Read the Serial Number back into a RAM buffer */
        E2PROM_E2Read( ADDRESS_OFFSET, abSerialNumberBuffer, NUMBER_OF_BYTES );
    }

    while(1)
    {
    }
```

```
}
```

The same code in assembly is:

```
;***************************************************************************
;    EEPROM User Module Example Code:
;
;    A SerialNumber EEPROM was created to start at block 250 with a length
;    of 10 bytes. Remember to edit the Flashsecurity.txt file to allow
;    writes to Flash (Set the 250th block to U – Unprotected).
;
;    This example:
;
;        a) Writes the initial data to the EEPROM block area
;        Note that this will invoke the SavePartial algorithm which
;        allocates a temporary 64-byte buffer on the stack. If
;        Flash memory is plentiful and the extra 54 bytes can be
;        wasted, set the SerialNumber device to a length of 64 and when
;        writing, specify a bytecount of 64. This will write an entire
;        block without using a temporary buffer. The extra 54 bytes
;        beyond the SerialNumber will be bogus data.
;
;        b) Reads the data back into a RAM buffer
;
;***************************************************************************
include "m8c.inc"       ; part specific constants and macros
include "memory.inc"    ; Constants and macros for SMM/LMM and Compiler
include "PSoCAPI.inc"   ; PSoC API definitions for all User Modules

ADDRESS_OFFSET:             equ     0
NUMBER_OF_BYTES:            equ     10
TEMPERATURE:                equ     25

export _main

export  abSerialNumberString
export  abInitialSerialNumber
export  abSerialNumberBuffer
export  bCounter
export  pPtr

    AREA    bss    (RAM,REL)

abInitialSerialNumber:    blk    NUMBER_OF_BYTES   ; string holds initial serial data
abSerialNumberBuffer:     blk    NUMBER_OF_BYTES   ; buffer to read back serial data
bCounter:                 blk     1                ; counter to load initial string
pPtr:                     blk     1                ; pointer to initial string

    AREA    text (ROM,REL)

;Table to hold initial serial number string
.LITERAL
abSerialNumberString:    db     '0','1','2','3','4','5','6','7','8','9'
.ENDLITERAL
```

```
_main:

; load the serialnumber into RAM
    mov    [bCounter], NUMBER_OF_BYTES       ; load 10 bytes from ROM into RAM

 RAM_SETPAGE_MVR >abInitialSerialNumber
 RAM_SETPAGE_MVW >abInitialSerialNumber
    mov    [pPtr], <abInitialSerialNumber ; ptr RAM data to put Flash data
    mov    X, <abSerialNumberString       ; LSB of abSerialNumberString
    mov    A, >abSerialNumberString          ; MSB of abSerialNumberString

; Use ROMX and MVI to copy the data from Flash to RAM
.loop:
    push   A                              ; Save MSB of abSerialNumberString to Stack
    romx
    mvi    [pPtr], A                      ; Save the value to RAM
    pop    A                              ; Get MSB of abSerialNumberString from Stack
    inc    X                              ; Increment LSB of abSerialNumberString
 dec    [bCounter]
 jnz    .loop


; Write the Serial Number - assume temp of 25C
    mov    A, TEMPERATURE                 ; temperature = 25C
    push   A
    mov    A, >NUMBER_OF_BYTES            ; MSB of wByteCount = 0
    push   A
    mov    A, <NUMBER_OF_BYTES        ; LSB of wByteCount = 10
    push   A
    mov    A, >abInitialSerialNumber      ; MSB of pbDest= >abInitialSerialNumber
    push   A
    mov    A, <abInitialSerialNumber  ; LSB of pbDest=abInitialSerialNumber
    push   A
    mov    A, >ADDRESS_OFFSET             ; MSB of wAddr=0
    push   A
    mov    A, <ADDRESS_OFFSET         ; LSB of wAddr=0
    push   A
    call   E2PROM_bE2Write                ; Write the data
    add    SP, -E2_WR_ARG_STACK_FRAME_SIZE
    pop    X


 ;if ( bError == NOERROR )
    cmp    A, 0
    jnz    .ExampleDone


 ; Read the Serial Number back into a RAM buffer
    mov    A, >NUMBER_OF_BYTES             ; MSB of wByteCount = 0
    push   A
    mov    A, <NUMBER_OF_BYTES         ; LSB of wByteCount = 10
    push   A
    mov    A, >abSerialNumberBuffer       ;MSB of pbDest= >abSerialNumberBuffer
    push   A
    mov    A, <abSerialNumberBuffer    ; LSB of pbDest=abInitialSerialNumber
```

```
    push   A
    mov    A, >ADDRESS_OFFSET             ; MSB of wAddr=0
    push   A
    mov    A, <ADDRESS_OFFSET           ; LSB of wAddr=0
    push   A
    call   E2PROM_E2Read
    add    SP, -E2_RD_ARG_STACK_FRAME_SIZE
    pop    X

.ExampleDone:
    jmp .ExampleDone
```

Version History

Version	Originator	Description
1.7	DHA	Added Version History table.
1.7.b	DHA	Updated assembler sample firmware source code.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2004-2013 Cypress Semiconductor Corporation. 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.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any 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.