PSoC® Creator™

System Reference Guide

Document Number: 001-53983, Rev. *A

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intl): 408.943.2600
http://www.cypress.com

Contents

1 Introduction

This document describes functions supplied by Cypress to give better access to chip resources. The functions are not part of the component libraries but may be used by them. The developer can use the function calls to reliably perform needed chip functions.

Functionality described in the document:

- Reading and Writing to Flash.
- Software DMA functionality.
- Controlling System Clocks.
- Power Management.
- Cache control.
- Delay functions.
- Global Interrupt control.
- Debugging and System faults.

Conventions

The following table lists the conventions used throughout this guide:

Convention	**Usage**
Courier New	Displays file locations and source code: `C:\ …cd\icc\`, user entered text
Italics	Displays file names and reference documentation: *sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures: [**Enter**] or [**Ctrl**] [**C**]
File > New Project	Represents menu paths: File > New Project > Clone
Bold	Displays commands, menu paths and selections, and icon names in procedures: Click the **Debugger** icon, and then click **Next**.
Text in gray boxes	Displays cautions or functionality unique to PSoC Creator or the PSoC device.

References

This guide is one of a set of documents pertaining to PSoC Creator and PSoC3. Refer to the following other documents as needed:

- PSoC Creator Online Help
- PSoC Creator Customization API Reference Guide
- PSoC3/5 Technical Reference Manual (TRM)

Revision History

Document Title: PSoC Creator Library Reference Guide
Document Number: Document # 001-53983

Revision	Date	Description of Change
**	10/5/09	New document.
*A	04/26/10	Updated CyDelay function to be independent of the compiler optimization setting.

2 Flash

Flash and EEPROM are programmed through the SPC (System Performance Controller) calls. The Flash/EEPROM specific API abstracts this for simplicity.

Some parts have ECC (Error correction codes) storage. This can be configured to be true ECC or used for configuration storage. If it is used for configuration storage the user must be aware when erasing sectors. No API is supplied for erasing a sector but can be created with SPC calls.If the configuration data is erased the hardware will not be configured on the next reset.

An alternative to erasing a sector is programming a row. A ROW (part specific size) of flash or EEPROM can be erased and programmed in one step. The functions provided will deal with configuration data if it exists and allow smaller granularity of erasing.

The caller must first call the CySetTemp and CySetFlashEEBuffer functions. The temperature is needed to adjust the write times to the flash for optimal performance. The Buffer is used to store intermediate data while communicating with the SPC. The SPC is push/pull with a register to send commands to and read data back from.

Flash or EEPROM can be written to one row at a time by calling the same function “CyWriteRowData”. The first parameter will determine the flash or EEPROM array. The number of Arrays that are flash and the number of Arrays that are EEPROM are specific to the exact part selected. The following lists the array IDs assigned to Flash and EEPROM.

- 0x00-0x3E : (63) Flash Arrays
- 0x3F : (1) Selects all Flash arrays(for parallel programming)
- 0x40-0x7F : (64) Embedded EEPROM Arrays

Check your part to know which array IDs are valid.

Hardware Configuration Options

A PSoC device can be selected that supports ECC or not. If a device supports ECC, it can be used for true Error Correction Codes or it can be used to store configuration data. If the ECC is being used for configuration data some are may be unused by configuration data and available to the user. Care must be taken not to lose the configuration data. The device will not function as configured by the user with out this data.

Application Programming Interface

The Application Programming Interface (API) routines are provided to give a strait forward approach to writing to the Flash and EEPROM.

cystatus CySetTemp(void)

Description: Executes an SPC command to get the temperature of the die. The other flash functions need the temperature to maximize the efficiency of the Flash and EEPROM write algorithms.

Parameters: None

Return Value: CYRET_SUCCESS if successful.
CYRET_LOCKED if the SPC is already in use.
CYRET_BAD_PARAM if the buffer is 0.

Side Effects: This operation will take anywhere from 2 – 80 ms to perform.

cystatus CySetFlashEEBuffer(uint8 * buffer)

Description: Sets the address of the temporary storage area for SPC commands used to write Flash and EEPROM. This buffer can be used for other things when the flash functions are not being used.

Parameters: uint8 * buffer – Address of block of memory to store temporary memory. The size of the block of memory is SIZEOF_FLASH_ROW + SIZEOF_ECC_ROW.

Return Value: CYRET_SUCCESS if successful.
CYRET_LOCKED if the SPC is already in use.
CYRET_BAD_PARAM if the buffer is 0.

Side Effects: None

cystatus CyWriteRowData(uint8 arrayId, uint16 rowAddress, uint8 rowData)

Description: Allows a row to be erased and programmed. If the array is a flash array and ECC is being used for configuration storage, the function will first read the ECC data in the row and concatenate the rowData. Erase the row and program the complete row to flash.

Parameters: uint8 arrayId – ID of the array that contains the sector to be erased.
Uint16 rowAddress – Row Number to erase and then program. 0, 1, 2, …
uint8 * rowData – Address of the data to be programmed. The size of this row will be SIZEOF_FLASH_ROW or SIZEOF_EEPROM_ROW depending on the 'arrayId'.
Note This cannot be the same buffer passed as SPC buffer

Return Value: CYRET_SUCCESS if successful.
CYRET_LOCKED if the SPC is already in use.
CYRET_UNKNOWN if there was an SPC error.

Side Effects: This operation could take up to two milliseconds to perform.

cystatus CyWriteRowConfig(uint8 arrayId, uint16 rowAddress, uint8 * rowConfig)

Description: If the device supports ECC and it is not being used for ECC, this function allows a row of configuration data (or user data) to be stored in the ECC area of flash only. The function will read the user data in the row and concatenate the rowConfig, erase the row, and program the complete row to flash.

Parameters: uint8 arrayId – ID of the array that contains the sector to be erased.
uint8 rowAddress – Row Number to erase and then program. 0, 1, 2, …
uint8 * rowConfig – Address of the data to be programmed. The size of this row will be equal to SIZEOF_ECC_ROW

Return Value: CYRET_SUCCESS if successful.
CYRET_LOCKED if the SPC is already in use.
CYRET_UNKNOWN if there was an SPC error.

Side Effects: None

void CyFlashEEActivePower(uint8 state)

Description: Enables or Disables the FLASH/EEPROM subsystems during the active power mode.

Parameters: uint8 state – 0 disables all other values enable.

Return Value: void.

Side Effects: None

void CyFlashEEStandbyPower(uint8 state)

Description: Enables or Disables the FLASH/EEPROM subsystems during the standby power mode.

Parameters: uint8 state – 0 disables all other values enable.

Return Value: void.

Side Effects: None

Defines

Define	Description
SIZEOF_ECC_ROW	Size of a row of configuration data.
SIZEOF_FLASH_ROW	Size of a row of Flash data.
SIZEOF_EEPROM_ROW	Size of a row of EEPROM data.

Sample Firmware Source Code

The following is a C language example demonstrating the basic functionality of the Flash.

```
#include <device.h>

/* Dont overwrite your own code. */
#define FLASH_START_TEST                     0x2000
#define ADDRESS_TO_ROW(A)               (A >>  8)

uint8 EccData1[SIZEOF_ECC_ROW] = {"User supplied data"};
uint8 Data1[SIZEOF_FLASH_ROW] = {"User supplied data"};
uint8 rowBuffer[SIZEOF_FLASH_ROW + SIZEOF_ECC_ROW];

void main()
{
    uint16 Index;
   cystatus Status;


   /* Must get temperature and set a temp buffer first. */
    if(CySetTemp() || CySetFlashEEBuffer(rowBuffer))


    /* Erase and Program the row. */
    Status = CyWriteRowData(0, ADDRESS_TO_ROW(FLASH_START_TEST), Data1);
   if(Status)
    {
        /* Handle Error. */
    }

    for(Index = 0; Index < SIZEOF_FLASH_ROW; Index++)
    {
        if(Data1[Index] != CY_GET_XTND_REG8((const void far *)
(CYDEV_FLS_BASE + FLASH_START_TEST + Index)))
        {
            /* Handle Error. */
        }
    }

#if (CYDEV_ECC_ENABLE == 0 && CYDEV_CONFIGURATION_ECC == 0)
    /* Erase and Program the row. */
    Status = CyWriteRowConfig(0, ADDRESS_TO_ROW(FLASH_START_TEST),
EccData1);
   if(Status)
    {
        /* Handle Error. */
    }

    for(Index = 0; Index < SIZEOF_ECC_ROW; Index++)
    {
        if(EccData1[Index] != CY_GET_XTND_REG8((const void far *)
(CYDEV_ECC_BASE + (FLASH_START_TEST >> 3)+ Index)))
        {
            /* Handle Error. */
        }
    }

#endif

    /***********************************************************
    *                                                          *
    * Power functions.                                         *
    *                                                          *
    ***********************************************************/

    /* Turn power off during active mode to the flash/eeprom block. */
    CyFlashEEActivePower(0);

    /* Turn power on during active mode to the flash/eeprom block. */
    CyFlashEEActivePower(1);

    /* Set Flash and EEPROM arrays to be active in standby mode. */
    CyFlashEEStandbyPower(1);

    /* Set Flash and EEPROM arrays to be disabled in standby mode. */
    CyFlashEEStandbyPower(0);

   while(2);
}
```

3 DMAC

The DMAC files provide the API functions for the DMA controller, DMA channels and Transfer Descriptors. This API is the library version not the auto generated code that is generated when the user places a DMA component on the schematic. The auto generated code would use the APIs in this module.

Note This code is endian agnostic.

The Transfer Descriptor memory can be used as standard memory (RAM) if the TDs are not being used.

This code uses the first byte of each TD to manage the free list of TDs. You will over write this once the TD is allocated for configuration purposes.

Application Programming Interface

The following are the DMAC APIs.

uint8 DMA_DmaInitialize(uint8 burstCount, uint8 requestPerBurst, uint16 upperSrcAddress, uint16 upperDestAddress)

Description: Allocates and initializes a DMAC channel to be used by the caller.

Parameters: (uint8) burstCount. Specifies the size of bursts (1 to 127) this TD should be divided into. If this value is zero then the whole transfer is done is one burst.

(uint8) requestPerBurst. A DMA request is required to activate each TD in a chain. If the TD requires multiple bursts to complete:

Value	Action
0	All subsequent bursts after the first burst will be automatically requested and carried out
1	All subsequent bursts after the first burst must also be individually requested.

(uint16) upperSrcAddress. Upper 16 bits of the source address.

(uint16) upperDestAddress. Upper 16 bits of the destination address.

Return Value: (uint8) The channel that can be used by the caller for DMA activity. DMA_INVALID_CHANNEL (0xFF) if there are no channels left.

Side Effects: None

void DMA_DmaRelease(void)

Description: Frees the channel associated with this instance of the component. The channel cannot be used again unless DMA_DmaInitialize is called again.

Parameters: None

Return Value: None

Side Effects: None

void CyDmacConfigure(void)

Description: Sets the value of the PHUB_CFG register with the define DMAC_DEFAULT_CONFIGURATION. The details of this register can be read in the TRM. The value of DMAC_DEFAULT_CONFIGURATION can be changed to meet the needs of the project.

Parameters: None

Return Value: None

Side Effects: None

uint8 CyDmacError(void)

Description: Return the value of the DMA_ERROR type, which contains the error types for the last failed DMA transaction.

Parameters: None

Return Value: Returns the error data (4 bits) from the DMA_ERROR type.

Bit	Define	Description
Bit 3	DMAC_PERIPH_ERR	Set to 1 when a peripheral responds to a bus transaction with an error response. Cleared by writing a 1.
Bit 2	DMAC_UNPOP_ACC	Set to 1 when an access is attempted to an invalid address. Cleared by writing a 1.
Bit 1	DMAC_BUS_TIMEOUT	Set to 1 when a bus timeout occurs. Cleared by writing a 1. Timeout values are determined by the BUS_TIMEOUT field in the PHUBCFG register.

Side Effects: None

void CyDmacClearError(uint8 error)

Description: Clears the error bits in the error register of the DMAC.

Parameters: uint8 error. Bitmask of the error bits to clear in the DMA_ERROR type.

Bit	Define	Description
Bit 3	DMAC_PERIPH_ERR	Set to 1 when a peripheral responds to a bus transaction with an error response. Cleared by writing a 1.
Bit 2	DMAC_UNPOP_ACC	Set to 1 when an access is attempted to an invalid address. Cleared by writing a 1.
Bit 1	DMAC_BUS_TIMEOUT	Set to 1 when a bus timeout occurs. Cleared by writing a 1. Timeout values are determined by the BUS_TIMEOUT field in the PHUBCFG register.

Return Value: None

Side Effects: None

uint32 CyDmacErrorAddress(void)

Description: When a BUS_TIMEOUT, UNPOP_ACC and PERIPH_ERR occurs the address of the error is written to the error address register and can be read with this function. If there are multiple errors, only the address of the first error is saved.

Parameters: None

Return Value: The address that caused the error.

Side Effects: None

uint8 CyDmaChAlloc(void)

Description: Allocates a channel from the DMAC to be used in all functions that require a channel handle.

Parameters: None

Return Value: The allocated channel number. Zero is a valid channel number. DMA_INVALID_CHANNEL is returned if there are no channels available.

Side Effects: None

cystatus CyDmaChFree(uint8 chHandle)

Description: Frees a channel handle allocated by CyDmaChAlloc.

Parameters: uint8 chHandle. The handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if chHandle is invalid.

Side Effects: None

cystatus CyDmaChEnable(uint8 chHandle, uint8 preserveTds)

Description: Enables the DMA channel. A software or hardware request still needs to happen before the channel will be executed.

Parameters: uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.

Parameters: uint8 preserveTds. Preserve the TDs transfer count, source and destination address's.

Value	Action
0	DMA controller should set Values of TDs to reflect current state of TD execution
1	DMA controller should not change values of the TD.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if chHandle is invalid.

Side Effects: None

cystatus CyDmaChDisable(uint8 chHandle)

Description: Disables the DMA channel. Once this function is called. CyDmaChStatus may be called to determine when the channel is disabled and determine which TDs were being executed.

Parameters: uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if chHandle is invalid.

Side Effects: None

cystatus CyDmaChPriority(uint8 chHandle, uint8 priority)

Description:	Sets the priority of a DMA channel.
Parameters:	uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.
	uint8 priority. The priority to set the channel to, 0 - 7.
Return Value:	CYRET_SUCCESS if successful.
	CYRET_BAD_PARAM if chHandle is invalid.
Side Effects:	None

cystatus CyDmaChSetExtendedAddress(uint8 chHandle, uint16 source, uint16 destination)

Description:	Sets the high 16 bits of the source and destination addresses for the DMA channel (all TD's in the chain).
Parameters:	uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.
	uint16 source. 16 bit address of the DMA transfer source.
	uint16 destination. 16 bit address of the DMA transfer destination.
Return Value:	CYRET_SUCCESS if successful.
	CYRET_BAD_PARAM if chHandle is invalid.
Side Effects:	None

cystatus CyDmaChSetInitialTd(uint8 chHandle, uint8 startTd)

Description:	Set the initial TD to be executed for the channel when the CyDmaChEnable function is called.
Parameters:	uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitialize.
	uint8 startTd. Index of TD to set as the first TD associated with the channel. Zero is a valid TD Index.
Return Value:	CYRET_SUCCESS if successful.
	CYRET_BAD_PARAM if chHandle is invalid.
Side Effects:	None

cystatus CyDmaChSetRequest(uint8 chHandle, uint8 request)

Description: Allows the caller to terminate a chain of TDs, terminate one TD, or create a direct request to start the DMA channel.

Parameters: uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.

uint8 request. One of the following constants. Each of the constants is a 3-bit value.

Request Values	Description
CPU_REQ	Create a direct request to start the DMA channel
CPU_TERM_TD	Terminate one TD
CPU_TERM_CHAIN	Terminate a chain of TDs

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if chHandle is invalid.

Side Effects: None

cystatus CyDmaChGetRequest(uint8 chHandle)

Description: This function allows the caller of CyDmaChSetRequest to determine if the request was completed.

Parameters: uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.

Return Value: Returns a 3-bit field corresponding to the 3 bits of the request that describes the state of the previously posted request. If a bit is zero, the request was completed.

DMA_INVALID_CHANNEL if the handle is invalid.

Side Effects: None

cystatus CyDmaChStatus(uint8 chHandle, uint8 * currentTd, uint8 * state)

Description: Determines the status of the current Transaction descriptor.

Parameters: uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitalize.

uint8 * currentTd. Address to store the Index of the current Transaction Descriptor. Can be NULL if the value is not needed.

uint8 * state. Address to store the State of the Channel. Can be NULL if the value is not needed.

Bit 1	STATUS_CHAIN_ACTIVE	0: channel is not currently being serviced by DMAC
		1: channel is currently being serviced by DMAC
Bit 0	STATUS_TD_ACTIVE	0: TD chain is inactive; either no DMA requests have triggered a new chain or the previous chain has completed.
		1: TD chain has been triggered by a DMA request

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if chHandle is invalid.

Side Effects: None

cystatus CyDmaChSetConfiguration(uint8 chHandle, uint8 burstCount, uint8 requestPerBurst, uint8 tdDone0, uint8 tdDone1, uint8 tdStop)

Description: Sets Configuration information for the channel.

Parameters: uint8 chHandle. A handle previously returned by CyDmaChAlloc or Dma_DmaInitialize.

uint8 burstCount. Specifies the size of small bursts (1 to 127) this TD should be divided into. If this value is zero then the whole transfer is done is one burst.

uint8 requestPerBurst. A DMA request is required to activate each TD in a chain. If the TD requires multiple bursts to complete:

Value	Action
0	All subsequent bursts after the first burst will be automatically requested and carried out
1	All subsequent bursts after the first burst must also be individually requested.

uint8 tdDone0. Selects one of the TERMOUT0 interrupt lines to signal completion. The line connected to the nrq terminal will determine the TERMOUT0_SEL definition and should be used as supplied by cyfitter.h

uint8 tdDone1. Selects one of the TERMOUT1 interrupt lines to signal completion. The line connected to the nrq terminal will determine the TERMOUT1_SEL definition and should be used as supplied by cyfitter.h

uint8 tdStop. Selects one of the TERMIN interrupt lines to signal to the DMAC that the TD should terminate.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if chHandle is invalid.

Side Effects: None

uint8 CyDmaTdAllocate(void)

Description: Allocates a Transaction Descriptor for use with an allocated DMA channel.

Parameters: None

Return Value: Zero based index of the Transaction Descriptor to be used by the caller. Zero is a valid TD index.

DMA_INVALID_TD if there are no free TDs available.

Side Effects: None

void CyDmaTdFree(uint8 tdHandle)

Description:	Returns a Transaction Descriptor to the free list.
Parameters:	uint8 tdHandle. Zero based index of the Transaction Descriptor to free.
Return Value:	None
Side Effects:	None

uint8 CyDmaTdFreeCount(void)

Description:	Returns the number of free Transaction Descriptors available to be allocated.
Parameters:	None
Return Value:	The number of free Transaction Descriptors.
Side Effects:	None

cystatus CyDmaTdSetConfiguration(uint8 tdHandle, uint16 transferCount, uint8 nextTd, uint8 configuration)

Description: Configures a Transaction Descriptor.

Parameters: uint8 tdHandle. A handle previously returned by CyDmaTdAlloc.

uint16 transferCount. Size of the data transfer (in bytes) for this Transaction Descriptor. Transfer count is limited to 0x0FFF. A larger value will cause the function to return CYRET_BADPARAM.

uint8 nextTd. Zero based index of the next Transaction Descriptor in the TD chain. Zero is a valid index to the next TD, DMA_INVALID_TD (0xFF) is end of chain.

uint8 configuration. Configuration bit field corresponding to bits 24 – 31 in the PHUB_TDMEMX_ORIG_TD0 register.

Configuration Options	Description
TD_SWAP_EN	Perform endian swap
TD_SWAP_SIZE4	Swap size = 4 bytes
TD_AUTO_EXEC_NEXT	The next TD in the chain will trigger automatically when the current TD completes
TD_TERMIN_EN	Terminate this TD if a positive edge on the "trq" input line occurs. The positive edge has to occur during a burst. That is the only time the DMAC will listen for it.
TD_TERMOUT1_EN	When this TD completes the TERMOUT1 signal selected by TERMOUT1_SEL will toggle if this bit is set.
TD_TERMOUT0_EN	When this TD completes the TERMOUT0 signal selected by TERMOUT0_SEL will toggle if this bit is set.
TD_INC_DST_ADR	Increment DST_ADR according to the size of each data transaction in the burst
TD_INC_SRC_ADR	Increment SRC_ADR according to the size of each data transaction in the burst

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if tdHandle is invalid.

Side Effects: None

cystatus CyDmaTdGetConfiguration(uint8 tdHandle, uint16 * transferCount, uint8 * nextTd, uint8 * configuration)

Description:	Retrieves the configuration of the Transaction Descriptor. If a NULL pointer is passed as a parameter, that parameter will be skipped. The user may request only the values they are interested in.
Parameters:	uint8 tdHandle. A handle previously returned by CyDmaTdAlloc.
	uint16 * transferCount. Address to store the size of the data transfer (in bytes) for this Transaction Descriptor (TD). Size of the data transfer (in bytes) for thisTD. A size of zero will cause the transfer to go indefinitely.
	uint8 * nextTd. Address to store the Zero based index of the next TD in the TD chain.
	uint8 * configuration. Address to store the Bit field of configuration bits. See CyDmaTdSetConfiguration.
Return Value:	CYRET_SUCCESS if successful.
	CYRET_BAD_PARAM if tdHandle is invalid.
Side Effects:	If a TD has a transfer count of N and is executed so the transfer count is 0 and then gets re-executed the Transfer count of zero will be interpreted as do forever. Be careful when requesting a td with a transfer count of zero.

cystatus CyDmaTdSetAddress(uint8 tdHandle, uint16 source, uint16 destination)

Description:	Sets the lower 16 bits of the source and destination addresses for this TD only.
Parameters:	uint8 tdHandle. A handle previously returned by CyDmaTdAlloc.
	uint16 source. Lower 16 address bits of the Source of the data transfer.
	uint16 destination. Lower 16 address bits of the Destination of the data transfer.
Return Value:	CYRET_SUCCESS if successful.
	CYRET_BAD_PARAM if tdHandle is invalid.
Side Effects:	None

cystatus CyDmaTdGetAddress(uint8 tdHandle, uint16 * source, uint16 * destination)

Description: Retrieves the lower 16 bits of the source and/or destination addresses for this TD only. if NULL is passed for a pointer parameter, that value will be skipped. The user may request only the values of interest.

Parameters: uint8 tdHandle. A handle previously returned by CyDmaTdAlloc.

uint16 * source. Address to store the lower 16 address bits of the Source of the data transfer.

uint16 * destination. Address to store the lower 16 address bits of the Destination of the data transfer.

Return Value: CYRET_SUCCESS if successful.

CYRET_BAD_PARAM if tdHandle is invalid.

Side Effects: None

Sample Firmware Source Code

The following is a C language example demonstrating the basic functionality of the DMA component. This example assumes the component has been placed in the schematic and renamed to MyDma.

An interrupt component named “MyDmaDone” has also been placed on the schematic and connected to the nrq terminal of the MyDma component.

```
#include <device.h>
#include <MyDma_dma.h>
#include <MyDmaDone.h>


uint8 Finished = 0;

CY_ISR(DmaDone)
{
    Finished = 1;
}

void main(void)
{
    uint8 MyTD;
    uint8 MyChannel;

    /* Perform dma in one burst.*/
    MyChannel = MyDma_DmaInitialize(0,
                                 0,   /* Automatically request carry out 
bursts.*/
                                 0,   /* upper address bits are zero. */
                                 0);  /* upper address bits are zero. */

    /* Get a Transaction Descriptor. */
    MyTD = CyDmaTdAllocate();
    if(MyTD == DMA_INVALID_TD)
    {
        /* Error Condition. */
    }


    /* Setup a TD. */

    /* Set TD to transfer 100 bytes with no next TD, */
    CyDmaTdSetConfiguration(MyTD,
                            100,
                            DMA_INVALID_TD,
                            TD_INC_DST_ADR | TD_INC_SRC_ADR |
TD_TERMOUT0_EN);

    /* Copy from 0x2000 to 0x3000. */
    CyDmaTdSetAddress(MyTD, 0x2000, 0x3000);

    /* Associate the TD with the channel. */
    CyDmaChSetInitialTd(MyChannel, MyTD);

    /* Setup the Interrupt connected to the nrq terminal. */
    MyDmaDone_SetVector(DmaDone);
    MyDmaDone_SetPriority(7);
    MyDmaDone_Enable();

    /* Enable the channel. */
    CyDmaChEnable(MyChannel, 0);

    /* Request DMA action. */
    CyDmaChSetRequest(MyChannel, CPU_REQ);

    /* Wait for the interrupt to signal completion. */
    while(!Finished)
        ;

    /* We are done with the DMA and Interrupt components. */
    MyDmaDone_Disable();
    MyDma_DmaRelease();
    while(1);
}
```

4 Library

The *CyLib.c* file contains the APIs for system clock, power management, memory, and string functions.

System Clock APIs

The following are the system clock APIs in the CyLib.c file:

void CyPLL_OUT_Start(void)

Description:	Enables PLL, and waits for it to become stable.
Parameters:	Void
Return Value:	Void
Side Effects:	The caller should know how the system clock tree is configured before starting and stopping the PLL. If it is driving other clocks this could be catastrophic.

void CyPLL_OUT_Stop(void)

Description:	Disables the PLL.
Parameters:	void
Return Value:	void
Side Effects:	The caller should know how the system clock tree is configured before starting and stopping the PLL. If it is driving other clocks this could be catastrophic.

void CyIMO_Start(void)

Description:	Enables the Internal Main Oscillator (IMO) by setting the enable bit in the 'Active Power Mode Configuration Register' corresponding to this clock.
Parameters:	Void
Return Value:	Void
Side Effects:	The caller should know how the system clock tree is configured before starting and stopping the IMO. If it is driving other clocks this could be catastrophic.

void CyIMO_Stop(void)

Description:	Disables the Internal Main Oscillator (IMO) by clearing the enable bit in the 'Active Power Mode Configuration Register' corresponding to this clock.
Parameters:	Void
Return Value:	Void
Side Effects:	The caller should know how the system clock tree is configured before starting and stopping the IMO. If it is driving other clocks this could be catastrophic.

void CyXTAL_Start(void)

Description:	Enables the 4 - 33 MHz crystal oscillator.
Parameters:	void
Return Value:	void
Side Effects:	The caller should know how the system clock tree is configured before starting and stopping the XMHZ clock. If it is driving other clocks this could be catastrophic.

void CyXTAL_Stop(void)

Description:	Disables the 4 - 33 MHz crystal oscillator.
Parameters:	void
Return Value:	void
Side Effects:	The caller should know how the system clock tree is configured before starting and stopping the XMHZ clock. If it is driving other clocks this could be catastrophic.

void CyXTAL_32KHZ_Start(void)

Description:	Enables the xternal 32k oscillator, and waits for it to become stable.
Parameters:	Void
Return Value:	Void
Side Effects:	None

void CyXTAL_32KHZ_Stop(void)

Description:	Disables the xternal 32k oscillator.
Parameters:	Void
Return Value:	Void
Side Effects:	None

void CyXTAL_32KHZ_SetPowerMode(uint8 mode)

Description:	Allows the caller to set the power mode for the external 32k oscillator. This setting only takes effect in sleep modes. During active modes, the oscillator always runs in high power mode.
Parameters:	uint8 mode. 0 = High Power Mode, 1 = Low Power Mode.
Return Value:	uint8. Previous power mode state.
Side Effects:	None

void ILO_SetPowerMode(uint8 mode)

Description:	Allows the caller to set the power mode for the Internal Low Speed Oscillator (ILO) clock.
Parameters:	uint8 mode. 0 = Faster start-up, internal bias left on, 1 = Slower start-up, internal bias off.
Return Value:	uint8. Previous power mode state.
Side Effects:	None

Using System Clocks APIs in your project

To include the functions in your project, you need to set the appropriate flag using the PSoC Creator Build Settings dialog.

1. In the Workspace Explorer, right-click on your project and select Build Settings...
2. When the dialog opens, expand Compiler in the tree and select Command Line.
3. On the right side in the Custom Flags field, enter the following as appropriate:

   For 8051 projects, enter: DEFINE (CYLIB_SYSTEM_CLOCKS=1)

   For Arm projects, enter: -D CYLIB_SYSTEM_CLOCKS=1

For the Keil compiler if you want to have multiple defines:

- DEFINE (CYLIB_POWER_MANAGEMENT=1,CYLIB_SYSTEM_CLOCKS=1)

For more information, see Build Settings in the PSoC Creator Help.

Power Management APIs

void CyWait(void)

Description:	Stops the CPU, but the part remains active. Exits wait state on receipt of a non-masked interrupt.
Parameters:	Void
Return Value:	Void
Side Effects:	None

void CyIdle(void)

Description:	Puts the part into idle mode. Exits idle state on receipt of a non-masked interrupt.
Parameters:	Void
Return Value:	Void
Side Effects:	None

void CySleep(void)

Description:	Puts the part into sleep mode. Exits sleep state on receipt of a non-masked interrupt.
Parameters:	Void
Return Value:	Void
Side Effects:	None

void CyHibernate(void)

Description:	Puts the part into hibernate mode. Exits hibernate state on receipt of a nonmasked port (I/O) interrupt (the other sources cannot wake from hibernate).
Parameters:	Void
Return Value:	Void
Side Effects:	None

Using Power Management APIs in your project

To include the functions in your project, you need to set the appropriate flag using the PSoC Creator Build Settings dialog.

1. In the Workspace Explorer, right-click on your project and select Build Settings...
2. When the dialog opens, expand Compiler in the tree and select Command Line.
3. On the right side in the Custom Flags field, enter the following as appropriate:
   For 8051 projects, enter: DEFINE (CYLIB_POWER_MANAGEMENT=1)
   For Arm projects, enter: -D CYLIB_SYSTEM_CLOCKS=1

For the Keil compiler if you want to have multiple defines:

- DEFINE (CYLIB_POWER_MANAGEMENT=1,CYLIB_SYSTEM_CLOCKS=1)

Other APIs

cystatus CyCacheLoadLockedLine(uint8 line, void * address)

Description:	Loads code into the cache SRAM and locks it. A line of code must be 64 bytes long and 64 byte aligned.
Parameters:	line: The line number in cache to load. address: Points to 64 bytes of code to load.
Return Value:	CYRET_SUCCESS or CYRET_BAD_PARAM.
Side Effects:	None

cystatus CyCacheUnlockLine(uint8 line)

Description:	Unlocks a line in cache.
Parameters:	line: The line number to unlock.
Return Value:	cystatus
Side Effects:	None

void CyDelay(uint32 milliseconds)

Description:	Delay by the specified number of milliseconds. By default the number of cycles to delay is calculated based on the clock configuration entered in PSoC Creator. If the clock configuration is changed at run-time, then the function CyDelayFreq is used to indicate the new Bus Clock frequency. CyDelay is used by several components, so changing the clock frequency without updating the frequency setting for the delay can cause those components to fail.
Parameters:	milliseconds: Number of milliseconds to delay.
Return Value:	Void
Side Effects:	None

void CyDelayUs(uint16 microseconds)

Description:	Delay by the specified number of microseconds. By default the number of cycles to delay is calculated based on the clock configuration entered in PSoC Creator. If the clock configuration is changed at run-time, then the function CyDelayFreq is used to indicate the new Bus Clock frequency. CyDelayUs is used by several components, so changing the clock frequency without updating the frequency setting for the delay can cause those components to fail.
Parameters:	microseconds: Number of microseconds to delay.
Return Value:	Void
Side Effects:	None

void CyDelayFreq(uint32 freq)

Description:	Sets the Bus Clock frequency used to calculate the number of cycles needed to implement a delay with CyDelay. By default the frequency used is based on the value determined by PSoC Creator at build time.
Parameters:	freq: Bus clock frequency in Hz.
Return Value:	Void
Side Effects:	None

void CyDelayCycles(uint32 cycles)

Description:	Delay by the specified number of cycles using a software delay loop.
Parameters:	cycles: Number of cycles to delay.
Return Value:	Void
Side Effects:	None

uint32 CyDisableInts(void)

Description:	Disables all interrupts.
Parameters:	Void
Return Value:	32 bit mask of previously enabled interrupts.
Side Effects:	None

uint32 CyEnableInts(uint32 intState)

Description:	Enables interrupts to a given state.
Parameters:	intState, 32 bit mask of interrupts to enable.
Return Value:	void
Side Effects:	None

void CySoftwareReset(void)

Description:	Performs a software reset.
Parameters:	void
Return Value:	void
Side Effects:	None

Global Interrupt Macros

CYGlobalIntEnable

Globally enables all interrupts that have been individually enabled. This will not affect interrupts that have been disabled.

CYGlobalIntDisable

Globally disables all interrupts that have been individually enabled. This will not change the enabled interrupts to disabled, requiring each individual bit to be re enabled, but globally allow the interrupts to be masked.

5 Pins

The *cypins.h* file contains the APIs to support port/pin access and control.

Application Programming Interface

The following are the physical pin APIs.

Note These APIs are always present and refer to the physical ports and pins on the device. The Pins Component Data Sheet contains the APIs generated for the Pins component. Those APIs will be similar to the physical pin APIs, but they will refer to the component instance name and not the physical pin. Use of the Pins component over using these APIs alone is strongly recommended. There is no performance cost for doing so and it comes with added safety and ease-of-migration for your design.

The pin APIs use a common parameter that is the address of the Pin Configuration register for the desired pin. PSoC Creator will generate generic #defines for all PSoC pins. Additional #defines are generated for pins that are defined and named using a Pins component.

The generic #defines for each PSoC pin are provided in the *cydevice_trm.h* file in the following form:

CYREG_PRTx_PCy	- x is the port number and y is pin number

uint8 CyPins_ReadPin(uint16/uint32 pinPC)

Description: Reads the pin configuration register of the specified pin and returns the state of the pin.

Parameters: uint16 (uint32) pinPC: address of the Pin Configuration register for the pin to be read. This parameter is a uint32 value for the PSoC 5 family of chips.

Return Value: uint8: the state of the I/O pin

0 –	the pin state is logic low or 0
non 0 –	the pin state is logic high or 1

Side Effects: None

void CyPins_SetPin(uint16/uint32 pinPC)

Description: Sets the pin state to logic high or 1.

Parameters: uint16 (uint32) pinPC: address of the Pin Configuration register for the pin to be set. This parameter is a uint32 value for the PSoC 5 family of chips.

Return Value: None

Side Effects: None

void CyPins_ClearPin(uint16/uint32 pinPC)

Description: Sets the pin state to logic low or 0.

Parameters: uint16 (uint32) pinPC: address of the Pin Configuration register for the pin to be cleared. This parameter is a uint32 value for the PSoC 5 family of chips.

Return Value: None

Side Effects: None

void CyPins_SetPinDriveMode(uint16/uint32 pinPC, uint8 mode)

Description: Sets the drive mode for the pin to one of eight supported modes.

Parameters: uint16 (uint32) pinPC: address of the Pin Configuration register for the pin. This parameter is a uint32 value for the PSoC 5 family of chips.

uint8 mode: desired drive mode for the pin

Mode Value	Notes
PIN_DM_ALG_HIZ	Analog HiZ
PIN_DM_DIG_HIZ	Digital HiZ
PIN_DM_RES_UP	Resistive pull up
PIN_DM_RES_DWN	Resistive pull down
PIN_DM_OD_LO	Open Drain – drive low
PIN_DM_OD_HI	Open Drain – drive high
PIN_DM_STRONG	Strong CMOS output
PIN_DM_RES_UPDWN	Resistive pull up/down

Return Value: None

Side Effects: None

uint8 CyPins_ReadPinDriveMode(uint16/uint32 pinPC)

Description: Reads the drive mode setting for the specified pin.

Parameters: uint16 (uint32) pinPC: address of the Pin Configuration register for the pin. This parameter is a uint32 value for the PSoC 5 family of chips.

Return Value: uint8 – returns the current drive mode setting.

Mode Value	Notes
PIN_DM_ALG_HIZ	Analog HiZ
PIN_DM_DIG_HIZ	Digital HiZ
PIN_DM_RES_UP	Resistive pull up
PIN_DM_RES_DWN	Resistive pull down
PIN_DM_OD_LO	Open Drain – drive low
PIN_DM_OD_HI	Open Drain – drive high
PIN_DM_STRONG	Strong CMOS output
PIN_DM_RES_UPDWN	Resistive pull up/down

Side Effects: None

Sample Firmware Source Code

The following is a C language example demonstrating the basic functionality of the pin API functions. The example assumes three port 0 pins are used. A Pins component is not associated with these pins. Port 0 pins 0 and 1 are used to control two LEDs and port 0, pin 4 is used as an input.

```
#include <device.h>
#include <cypins.h>

void main()
{
    uint8 inputState;

    /* Port 0, pin 4 is an input */
    /* Set mode; Set initial condition to high */
    CyPins_SetPinMode(CYREG_PRT0_PC4,PIN_DM_RES_UP );
    CyPins_SetPin(CYREG_PRT0_PC4);
    /* Set drive mode for LED pins */
    CyPins_SetPinDriveMode(CYREG_PRT0_PC0, PIN_DM_STRONG);
    CyPins_SetPinDriveMode(CYREG_PRT0_PC1, PIN_DM_STRONG);
    /* turn one LED off, one on */
    CyPins_ClearPin(CYREG_PRT0_PC0);
    CyPins_SetPin(CYREG_PRT0_PC1);
    /* Read input pin */
    inputState = CyPins_ReadPin(CYREG_PRT0_PC4);
}
```

6 Boot Loader System

In PSoC Creator, the Boot Loader system manages the process of updating the PSoC 3/5 device flash memory with new application code and/or data ("code"). This is accomplished by the following parts:

- Bootloader component
- Communications component
- Boot Loader project, used to create the Bootloader component
- Bootloadable project, used to create the code

The following sections describe the various aspects of the Boot Loader process in greater detail.

Bootloader Component

The Bootloader component allows you to update the PSoC3/5 device flash memory with new CODE. The Bootloader accepts and executes commands, and passes responses to those commands back to the communications component. The Bootloader collects and arranges the received data and manages the actual writing of flash through a simple command/status register interface. The Bootloader component is not a typical component. It is not available in the Component Catalog. Instead it is always present, behind the scenes, if you have a Bootloader type project.

Communications Component

The Communications component manages the communications protocol to receive commands from an external system, and passes those commands to the Bootloader. It also passes command responses from the Bootloader back to the off-chip system.

Note The I^2C is the only supported communication method for the bootloader. Also, you must select the hardware I^2C, not UDB-based I^2C.

Boot Loader Project Types

In order to implement both a Bootloader component and the code, you must create special PSoC Creator project types: Boot Loader Project and Bootloadable Project.

Boot Loader Project

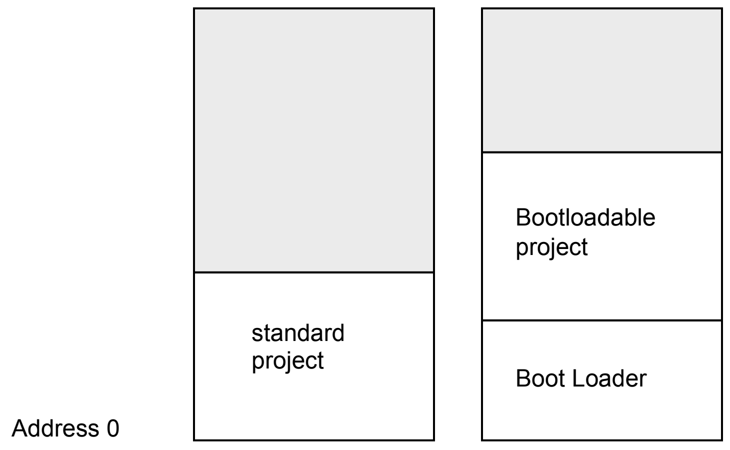
The Bootloader component only exists within a PSoC Creator design project of type “Boot Loader.” When you create a Boot Loader project, a Bootloader component automatically exists in the project.

You typically complete a Boot Loader design project by dragging a communications component onto the schematic, routing the I/O to pins, setting up clocks, etc. A Boot Loader project with a communications component implements the basic boot loader function of receiving new code and writing it to flash. You can add custom functions to a basic Boot Loader project by dragging other components onto the schematic or by adding source code.

Bootloadable Project

The Bootloadable project is actually the code. It is very similar to a “standard” design project; the project type can easily be changed between the two during the design phase. The main differences are that a Bootloadable project is always associated with a Boot Loader project, and a standard project is never associated with a Boot Loader project.

A standard project resides in flash starting at address zero. A Bootloadable project occupies flash at an address above zero; the associated Boot Loader project occupies flash starting at address zero, as shown in the following:



Boot Loader and Bootloadable Project Functions

The Boot Loader project performs overall transfer of a Bootloadable project, or new code, to the flash via the Boot Loader project’s communications component. After the transfer, the processor is always reset. The Boot Loader project is also responsible at reset time for testing certain conditions and possibly auto-initiating a transfer if the Boot Loadable project is non-existent or is corrupt.

At startup, the Boot Loader code loads configuration bytes for its own configuration. It must also initialize the stack and other resources and peripherals to do the transfer. When the transfer is complete, control is passed to the Bootloadable project via a software reset.

The Bootloadable project then loads configuration bytes for its own configuration; and re-initializes the stack and other resources and peripherals for its functions. The Bootloadable project may call the CyBtldr_Load() function in the Boot Loader project to initiate a transfer (this results in another software reset).

PSoC Creator Project Output Files

When either project type – Boot Loader or Bootloadable - is built, an output file is created for that project.

In addition, an output file for both projects – a “combination” file – is created when the Bootloadable project is build. This file includes both the Boot Loader and Bootloadable projects. This file is typically used to facilitate downloading both projects (via JTAG / SWD) to device flash in a production environment.

For Boot Loader projects the configuration bytes are always stored in the main flash that is occupied by the Boot Loader, and never in ECC flash.

Configuration bytes for Bootloadable projects may be stored in either main flash or in ECC flash. The format of the Bootloadable project output file is such that when the device has ECC bytes which are disabled, transfer operations are executed in less time. This is done by interleaving records in the Bootloadable main flash address space with records in the ECC flash address space. The Bootloader takes advantage of this interleaved structure by programming the associated flash row once – the row contains bytes for both main flash and ECC flash.

Each project has its own checksum. The checksums is included in the output files at project build time.

Memory Usage

The Boot Loader project always occupies the bottom N 256-byte blocks of flash. N is set so that there is enough flash for:

- the vector table for this project, starting at address 0 (PSoC5 only), and
- the Boot Loader project configuration bytes, and
- the Boot Loader project code and data, and
- the checksum for the Boot Loader portion of flash.

Note that the Boot Loader project configuration bytes are always stored in main flash, never in ECC flash. The relevant option is removed from the project's .cydwr file.

The Boot Loader portion of flash is protected; it can only be overwritten by downloading via JTAG / SWD.

The Bootloadable project occupies flash starting at the first 256-byte boundary after the Boot Loader, and includes:

- the vector table for the project (PSoC5 only), and
- the Bootloadable project code and data.

The Bootloadable project's configuration bytes may be stored in the same manner as in a standard project, i.e. in either main flash or in ECC flash, per settings in the project's .cydwr file.

The highest 64-byte block of flash is used as a common area for both projects. Various parameters are saved in this block, which may include:

- the entry in flash of the Bootloadable project (4 byte address)
- the amount of flash occupied by the Bootloadable project (Number of flash rows)
- the checksum for the Bootloadable portion of flash (one byte)
- the size of the Bootloadable portion of flash (4 bytes, in bytes)

PSoC3 8051 Details

In the PSoC3, the only "exception vector" is the 3-byte instruction at address 0, which is executed at processor reset. (The interrupt vectors are not in flash – they are supplied by the Interrupt Controller (IC)). So at reset the 8051 Boot Loader code simply starts executing from flash address 0.

PSoC5 ARM Cortex-M3 Details

In the PSoC5, a table of exception vectors must exist at address 0. (The table is pointed to by the Vector Table Offset Register, at address 0xE000ED08, whose value is set to 0 at reset.) The Boot Loader code starts immediately after this table.

The table contains the initial stack pointer (SP) value for the Boot Loader project, and the address of the start of the Boot Loader project code. It also contains vectors for the exceptions and interrupts to be used by the Boot Loader.

The Bootloadable project also has its own vector table, which contains that project's starting SP value and first instruction address. When the transfer is complete, as part of passing control to the Bootloadable project the value in the Vector Table Offset Register is changed to the address of the Bootloadable project's table.

Bootloader Parameters

To access the Bootloader parameters, open the System DWR and expand the Bootloader section of the editor.

Wait for Command

Description:	At reset, if the Bootloader detects that the checksum in Bootloadable project flash is valid then it may optionally wait for a command to start a transfer operation before jumping to the Bootloadable project code.
Settings:	Yes or no that the wait will take place.
Default:	Yes
Modifiable by API or Driver Firmware:	No
Relationship to other parameters	If the selection is "yes" then the Wait for Command Time parameter is editable. If the selection is "no" then that parameter is grayed out. In that case an external system typically is not able to initiate a transfer, however the Bootloadable project code can still launch a transfer operation by calling Bootloader_Start().

Wait for Command Time

Description:	At reset, if the Bootloader detects that the checksum in Bootloadable project flash is valid then it may optionally wait for a command to start a transfer operation before jumping to the Bootloadable project code. This parameter is the wait timeout period.
Settings:	1 – 255, in units of 10 msec.
Default:	10, or 100 msec.
Modifiable by API or Driver Firmware:	No
Relationship to other parameters	This parameter is editable only if the Wait for Command parameter is set to "yes", otherwise it is grayed out.

IO Component

Description:	This is the communications component that the Bootloader uses to receive commands and send responses. One and only one communications Component must be selected. Only two-way communications components are used, e.g. a UART must have both RX and TX enabled, and an infrared (IrDA) component could not be used. A design rule check (DRC) exists for the case where no two-way communications component has been placed onto the Boot Loader project schematic.
Settings:	This property is a list of the available IO communications protocols on the schematic that have bootloader support. There is typically only one communications Component on a Boot Loader project schematic, but there may be more in the case where the Boot Loader must also do a custom function during the transfer.
Default:	If only one communications Component is on the schematic then that Component is the only component available in the DWR dropdown.
Modifiable by API or Driver Firmware:	No
Relationship to other parameters	None.

Bootloader API

The Bootloader provides a public API call solely for the purpose of launching a transfer operation from Bootloadable project code. Once called, a software reset is executed, and then the Bootloader takes over the CPU. Bootloadable project code, including interrupt handlers, is not executed.

When a transfer operation starts, resources and peripherals are reconfigured as needed. All other resources and peripherals are disabled.

When the transfer operation is complete, the CPU is reset.

void CyBtldr_Load(void)

Description:	Starts a transfer operation. Reconfigures the device per Boot Loader project configuration.
Parameters:	void
Return Value:	None. The processor is reset when the transfer is complete.
Side Effects:	None

Bootloader Commands

The Bootloader supports the following commands. All received bytes that do not start with one of the set of command bytes listed below is discarded with no response generated. All multi byte fields are output LSB-first.

Enter Bootloader

All other commands are ignored until this command is received.

Input

- Command Byte: 0x38
- Data Bytes: NA

Output

- Status/Error Codes:

  Success
  Error Command
  Error Packet Data
  Error Packet Checksum

- Data Bytes:

  4 bytes - Silicon ID
  1 byte - Silicon Rev
  3 bytes - Bootloader Version

Get Flash Size

Responds with the first and last available rows in the selected flash array.

Input

- Command Byte: 0x32
- Data Bytes:

  1 byte - Flash Array ID

Output

- Status/Error Codes:

  Success
  Error Command
  Error Packet Data
  Error Packet Checksum
  Error Packet Length
  Error Flash Array

- Data Bytes:

  2 bytes - First available row

2 bytes - Last available row

Program Row

Writes one row of flash data to the device.

Input

- Command Byte: 0x39
- Data Bytes:

  1 byte - Flash Array ID
  2 bytes - Flash Row Number
  n bytes - Data to write into the flash row

Output

- Status/Error Codes:

  Success
  Error Command
  Error Packet Data
  Error Packet Checksum
  Error Packet Length
  Error Flash Array
  Error Flash Checksum

- Data Bytes: NA

Erase Row

Erases the contents of the provided flash row.

Input

- Command Byte: 0x34
- Data Bytes:

  1 byte - Flash Array ID
  2 bytes - Flash Row Number

Output

- Status/Error Codes:

  Success
  Error Command
  Error Packet Data
  Error Packet Checksum
  Error Packet Length
  Error Flash Array

- Data Bytes: NA

Verify Row

Gets a 1 byte checksum for the contents of the provided row of flash

Input

- Command Byte: 0x3A
- Data Bytes:

  1 byte - Flash Array ID
  2 bytes - Flash Row Number

Output

- Status/Error Codes:

  Success
  Error Command
  Error Packet Data
  Error Packet Checksum
  Error Packet Length
  Error Flash Array

- Data Bytes:

  1 byte - Row checksum

Verify Checksum

Gets a 1 byte checksum for the contents of the entire flash

Input

- Command Byte: 0x31
- Data Bytes: NA

Output

- Status/Error Codes:

  Success
  Error Command
  Error Packet Data
  Error Packet Checksum
  Error Packet Length
  Error Verify Image
  Error Flash Checksum

- Data Bytes:

  1 byte - Flash checksum

Send Data

Sends a block of data to the device. This data is buffered up in anticipation of another command that will inform the bootloader what to do with the data. If multiple send data commands are issued back-to-back, the data is appended to the previous block. This command is used to breakup large transfers into smaller pieces to prevent bus starvation in some protocols.

Input

- Command Byte: 0x37
- Data Bytes: n bytes - Data to save in the device

Output

- Status/Error Codes:

  Success
  Error Command
  Error Packet Data
  Error Packet Checksum
  Error Packet Length
- Data Bytes: NA

Sync Boot Loader

Resets the bootloader to a clean state, ready to accept a new command. Any data that was buffered will be thrown out. This is only necessary if the host and client get out of sync with each other.

Input

- Command Byte: 0x35
- Data Bytes: NA

Output

- NA - This packet is not acknowledged

Exit Bootloader

This stops the booting process and triggers the device to reset itself.

Input

- Command Byte: 0x3B
- Data Bytes: NA

Output

- NA - This packet is not acknowledged

Bootloader Packets

Packets sent to the Bootloader have the following structure:

- 1-byte packet start (0x01)
- 1-byte command
- 2-bytes data length
- n-bytes data
- 2-bytes checksum
- 1-byte packet end (0x17)

Packets output from the Bootloader have the following structure:

- 1-byte packet start (0x01)
- 1-byte status/error code
- 2-bytes data length
- n-bytes data
- 2-bytes checksum
- 1-byte packet end (0x17)

Bootloader Status/Error Codes

The possible status/error codes output from the bootloader are:

Success

The command was successfully received and executed. Value = 0x00

Error Command

The command ID is invalid. Value = 0x01

Error Flash Array

The provided flash array is invalid. Value = 0x02

Error Packet Data

The provided packet data is invalid. Value = 0x03

Error Packet Length

The packet's length does not match what the bootloader expected. Value = 0x04

Error Packet Checksum

The packet's checksum does not match the contents. Value = 0x05

Error Flash Protection

The flash protection settings are invalid. Value = 0x06

Error Flash Checksum

The checksum value in the flash is invalid. Value = 0x07

Error Verify Image

The flash image is invalid. Value = 0x08

Bootloader Host Source Code

PSoC Creator ships with some C source code that can be used in the development of a bootloader host application. The host is the application that communicates with the bootloader itself to send new bootloadable images. The source code is located in the following directory:

<install directory>\cybootloaderutils\

By default, this directory is:

C:\Program Files\Cypress\PSoC Creator\1.0\PSoC Creator\cybootloaderutils\

This source code can be used to handle parsing of the *.cyacd files used to contain the bootloadable images, and to create packets used to communicate with the bootloader itself.

7 cy_boot Component Changes

This section lists and describes the major changes in the cy_boot component for each version:

Version	Description of Changes	Reason for Changes / Impact
1.40	Updated and changed the CyDelay function to be independent of the compiler optimization setting. Added a function named CyDelayUs.	The length of the delay varied when the selected compiler or optimization settings were changed. In some configurations, such as GCC with optimizations enabled, the delay loop was removed by the compiler. These variations resulted in a delay that was too short. The updated CyDelay function uses an assembly language subroutine called CyDelayCycles, which is independent of compiler optimizations. If the firmware modifies the bus clock speed, it must call CyDelayFreq to ensure that CyDelay will wait for the correct amount of time
	Updated cybtldr.c to prevent critical items from being optimized out	Optimizations performed by the ARM tool chains were causing critical items to be optimized out of the bootloader. This could cause the bootloader not to function correctly. These items have been modified to prevent the tool chains from removing them.