UM1618
User manual

Standard software driver for C55 Flash

Introduction

This document is the user manual for the Standard Software Driver (SSD) for single C55 Flash module.

The SSD is a set of APIs that enables user application to operate on the Flash module embedded on a microcontroller. The C55 SSD contains a set of functions to program/erase a single C55 Flash module.

The C55 Standard Software Driver (SSD) Flash provides the following APIs:

- FlashInit
- FlashErase
- FlashEraseAlternate
- BlankCheck
- FlashProgram
- ProgramVerify
- CheckSum
- FlashCheckStatus
- FlashSuspend
- FlashResume
- GetLock
- SetLock
- OverPgmProtGetStatus
- FlashArrayIntegrityCheck
- FlashArrayIntegritySuspend
- FlashArrayIntegrityResume
- UserMarginReadCheck

Contents

List of tables

1 Introduction

1.1 Document overview

The roadmap for the document is as follows:

Section 1.2 shows the features of the driver. *Appendix C: System requirements* details the system requirement for the driver development. *Appendix E: Document references* and lists the documents referred and terms used in making of this document. *Appendix D: Acronyms* lists the acronyms used.

Chapter 2 describes the API specifications. In this section there are many sub sections, which describe the different aspects of the driver. *Section 2.1* provides a general overview of the driver. *Section 2.2* mentions about the type definitions used for the driver. *Section 2.3* mentions the driver configuration parameters. *Section 2.4* and *Section 2.5* describe the data context structure and some other data structures used in this driver. *Section 2.6* provides return code information. *Section 2.7* and *Section 2.8* provide the detailed description of normal mode and user's test mode standard software Flash Driver APIs' respectively.

1.2 Features

The C55 SSD provides the following features:

- Driver binary built with Variable-Length-Encoding (VLE) instruction set.
- Driver released in binary c-array format to provide compiler-independent support for non-debug-mode embedded applications.
- Driver released in s-record format to provide compiler-independent support for debug-mode/JTAG programming tools.
- Each driver function is independent of each other so the end user can choose the function subset to meet their particular needs.
- Support from word-wise to quad-page-wise programming according to specific hardware feature for fast programming.
- Position-independent and ROM-able
- Ready-to-use demos illustrating the usage of the driver
- Concurrency support via asynchronous design.

2 API specification

2.1 General overview

The C55 SSD has APIs to handle the erase, program, erase verify and program verify operations on the Flash. Apart from these, it also provides the feature for locking specific blocks and calculating check sum. This SSD also provides four User Test APIs for checking the Array Integrity and do user margin read check as well as do suspend/resume those operations. All functions work as asynchronous model for concurrency event support by invoking '*FlashCheckStatus'* function to track the on-going status of targeted operation.

2.2 General type definitions

Table 1. Type definitions

Derived type	Size	C language type description
BOOL	8-bits	unsigned char
INT8	8-bits	signed char
VINT8	8-bits	volatile signed char
UINT8	8-bits	unsigned char
VUINT8	8-bits	volatile unsigned char
INT16	16-bits	signed short
VINT16	16-bits	volatile signed short
UINT16	16-bits	unsigned short
VUINT16	16-bits	volatile unsigned short
INT32	32-bits	signed long
VINT32	32-bits	volatile signed long
UINT32	32-bits	unsigned long
VUINT32	32-bits	volatile unsigned long

2.3 SSD configuration parameter

The configuration parameter which is used for SSD operations is explained in this section. The configuration parameters are handled as structure. User should correctly initialize the fields including *c55RegBase*, *mainArrayBase*, *uTestArrayBase*,, *mainInterfaceFlag*, *programmableSize* and *BDMEnable* before passing the structure to SSD functions. The rest of parameters such as *lowBlockInfo*, *midBlockInfo, highBlockInfo* and *nLargeBlockNum*, are initialized by '*FlashInit'* automatically and can be used for other purposes of user's application.

Table 2. SSD configuration structure field definition

Parameter name	Type	Parameter description
c55RegBase	UINT32	The base address of C55 control registers.
mainArrayBase	UINT32	The base address of Flash main array.
lowBlockInfo	BLOCK_INFO	Block info of the low address space. It includes information of this block space based on different block sizes.
midBlockInfo	BLOCK_INFO	Block info of the mid address space. It includes information of this block space based on different block sizes.
highBlockInfo	BLOCK_INFO	Block info of the high address space. It includes information of this block space based on different block sizes.
nLargeBlockNum	UINT32	Number of blocks of the large address space (128K or 256K).
uTestArrayBase	UINT32	The base address of the UTest block.
mainInterfaceFlag	BOOL	The flag to select main interface or not.
programmableSize	UINT32	The maximum programmable size of the C55 Flash according to specific interface.
BDMEnable	BOOL	The debug mode selection. User can enable/disable debug mode via this input argument.

The type definition for the structure is given below.

```
typedef struct _c55_ssd_config
{
UINT32 c55RegBase;
UINT32 mainArrayBase;
BLOCK_INFO lowBlockInfo;
BLOCK_INFO midBlockInfo;
BLOCK_INFO highBlockInfo;
UINT32 nLargeBlockNum;
UINT32 n8KBlockNum;
UINT32 uTestArrayBase;
BOOL mainInterfaceFlag;
UINT32 programmableSize;
BOOL BDMEnable;
} SSD_CONFIG, *PSSD_CONFIG;
```

2.4 Context data structure

The Context Data structure is used for storing the context variable values while an operation is in-progress. The operations that support asynchronous model may require caching the context data including '*FlashProgram*', '*ProgramVerify*', '*BlankCheck*', '*CheckSum*', '*FlashArrayIntegrityCheck*', and

'*UserMarginReadCheck*'. User needs to declare and initialize a context data structure before passing it to the above SSD functions. Refer to '*FlashCheckStatus*' to have a quick view of how to initialize the context data. The context data structure contents can be reviewed at any time during the operation progress (these information may be useful in some cases), but they must not be changed for any reason in order to make the operation completes correctly.

Table 3. Context data structure field definitions

Name	Description
dest	The context destination address of an operation
size	The context size of an operation
source	The context source of an operation
pFailedAddress	The context failed address of an operation
pFailedData	The context failed data of an operation
pFailedSource	The context failed source of an operation
pSum	The context sum of an operation
pMisr	The context MISR values of an operation
pReqCompletionFn	Function pointer to the Flash function being checked for status

The type definition for the structure is given below.

```
typedef struct _c55_context_data
{
UINT32 dest;
UINT32 size;
UINT32 source;
UINT32 *pFailedAddress;
UINT32 *pFailedData;
UINT32 *pFailedSource;
UINT32 *pSum;
MISR *pMisr;
void* pReqCompletionFn;
} CONTEXT_DATA, *PCONTEXT_DATA;
```

2.5 Other data structures

Some other data structures used for SSD operation is explained in this section. They are the structures used for variable declaration in SSD configuration and context data structures or input argument declaration in some APIs.

Table 4. Block information structure field definitions

Name	Type	Definition
n8KBlockNum	UINT32	Number of 8K block.
n16KBlockNum	UINT32	Number of 16K block.
n32KBlockNum	UINT32	Number of 32K block.
n64KBlockNum	UINT32	Number of 64K block.

The type definition for the structure is given below:

```
typedef struct _c55_block_info
{
UINT32 n8KBlockNum;
UINT32 n16KBlockNum;
UINT32 n32KBlockNum;
UINT32 n64KBlockNum;
} BLOCK_INFO, *PBLOCK_INFO;
```

Table 5. Large block select structure field definitions

Name	Type	Definition
firstLargeBlockSelect	UINT32	Bit map for the first 32 bit block select (from bit 0 to bit 31) in Large block (128K or 256K block) space such that bit 0 is corresponding to the least significant bit and bit 31 is corresponding to the most significant bit.
secondLargeBlockSelect	UINT32	Bit map for the second 32 bit block select (from bit 32 to upper bits) in Large block (128K or 256K block) space such that bit 32 is corresponding to the least significant bit and bit 63 is corresponding to the most significant bit.

The type definition for the structure is given below:

```
typedef struct _c55_nLarge_block_sel
{
UINT32 firstLargeBlockSelect;
UINT32 secondLargeBlockSelect;
} NLARGE_BLOCK_SEL, *PNLARGE_BLOCK_SEL;
```

Table 6. MISR structure field definitions

Name	Type	Definition
Wn n = 0, 1, …9	UINT32	Each Wn is corresponding to each MISR value provided by user. User must provide totally ten MISR values via this structure to do user's test mode functions.

The type definition for the structure is given below:

```
typedef struct _c55_misr
{
UINT32 w0;
UINT32 w1;
UINT32 w2;
UINT32 w3;
UINT32 w4;
UINT32 w5;
UINT32 w6;
UINT32 w7;
UINT32 w8;
UINT32 w9;
} MISR, *PMISR;
```

2.6 Return codes

The return code is returned to the caller function to notify the success or errors of the API execution. These are the possible values of return code:

Table 7. Return codes

Name	Value	Description
C55_OK	0x00000000	The requested operation is successful.
C55_ERROR_ALIGNMENT	0x00000001	Alignment error.
C55_ERROR_BUSY	0x00000004	New program/erase cannot be performed while a high voltage operation is already in progress. New array integrity cannot be performed while an array integrity is going on.
C55_ERROR_PGOOD	0x00000008	The program operation is unsuccessful.
C55_ERROR_EGOOD	0x00000010	The erase operation is unsuccessful.
C55_ERROR_NOT_BLANK	0x00000020	There is a non-blank Flash memory location within the checked Flash memory region.
C55_ERROR_VERIFY	0x00000040	There is a mismatch between the source data and the content in the checked Flash memory.
C55_ERROR_BLOCK_INDICATOR	0x00000080	Invalid block space indicator.
C55_ERROR_ALTERNATE	0x00000100	The operation does not support alternate interface for the specified address space.
C55_ERROR_FACTORY_OP	0x00000200	Factory erase/program is locked.
C55_ERROR_MISMATCH	0x00000400	In '*FlashArrayIntegrityCheck*' or '*UserMarginReadCheck*', the MISR values generated by the hardware do not match the values passed by the user.

Table 7. Return codes (continued)

Name	Value	Description
C55_ERROR_NO_BLOCK	0x00000800	In '*FlashArrayIntegrityCheck' or 'UserMarginReadCheck*', no block has been enabled for array integrity check.
C55_ERROR_ADDR_SEQ	0x00001000	Invalid address sequence error.
C55_ERROR_MARGIN_LEVEL	0x00002000	Invalid margin level error.
C55_DONE	0x00010000	The operation has been done and there is no more this operation requested on FlashCheckStatus function.
C55_INPROGRESS	0x00020000	The operation is in progress and user need call FlashCheckStatus more times finish this operation.

2.7 Normal mode functions

2.7.1 FlashInit

Description

This function initializes an individual Flash module. It accesses to Flash configuration register and read out the number of block for each memory space of single Flash module.

For each time of using this driver, user must provide the chip-dependent parameters such as *c55RegBase, mainArrayBase, uTestArrayBase, mainInterfaceFlag, programmableSize and DBMEnable* and the rest of parameters are initialized via this function. Those are block information including number of block based on block size for each address space.

Prototype

```
UINT32 FlashInit (PSSD_CONFIG pSSDConfig);
```

Arguments

Table 8. Arguments for FlashInit

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.

Return values

Table 9. Return values for FlashInit

Type	Description	Possible values
UINT32	Indicates successful completion of operation.	C55_OK

Troubleshooting

None.

Comments

In case of *mainInterfaceFlag* is main interface, '*FlashInit*' checks the C55_MCR_RWE, C55_MCR_EER and C55_MCR_SBC bits, and then clear them if any of them is set.

This function also clears PGM/ERS bit in MCR/MCRA register if any of them is set.

Assumptions

None.

2.7.2 FlashErase

Description

This function is to do erase operation for multi-blocks on single Flash module according to user's input arguments via main interface. The targeted Flash module status is checked in advance to return relevant error code if any. This function only sets the high voltage without waiting for the operation to be finished. Instead, user must call '*FlashCheckStatus*' function to confirm the successful completion of this operation.

Prototype

```
UINT32 FlashErase(PSSD_CONFIG pSSDConfig,
UINT8 eraseOption,
UINT32 lowBlockSelect,
UINT32 midBlockSelect,
UINT32 highBlockSelect,
NLARGEK_BLOCK_SEL nLargeBlockSelect);
```

Arguments

Table 10. Arguments for FlashErase

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
eraseOption	The option is to select user's expected erase operation.	The valid value can be: C55_ERASE_MAIN (0x0) C55_ERASE_MAIN_FERS (0x1) C55_ERASE_UTEST (0x2) C55_ERASE_UTEST_FERS (0x3)

Table 10. Arguments for FlashErase (continued)

Argument	Description	Range
lowBlockSelect	To select the array blocks in low address space for erasing.	Bit-mapped value such that the least significant bit is at bit 0 of 16K block region (if available), then 32K block region (if available) and lastly 64K block region (if available). Select the block in the low address space to be erased by setting 1 to the appropriate bit of *lowBlockSelect*. If there is not any block to be erased in the low address space, *lowBlockSelect* must be set to 0.
midBlockSelect	To select the array blocks in mid address space for erasing.	Bit-mapped value such that the least significant bit is at bit 0 of 16K block region (if available), then 32K block region (if available) and lastly 64K block region (if available). Select the block in the middle address space to be erased by setting 1 to the appropriate bit of *midBlockSelect*. If there is not any block to be erased in the middle address space, *midBlockSelect* must be set to 0.
highBlockSelect	To select the array blocks in high address space for erasing.	Bit-mapped value such that the least significant bit is at bit 0 of 16K block region (if available), then 32K block region (if available) and lastly 64K block region (if available). Select the block in the high address space to be erased by setting 1 to the appropriate bit of *highBlockSelect*. If there is not any block to be erased in the high address space, *highBlockSelect* must be set to 0.
nLargeBlockSelect	To select the array blocks in Large (128K or 256K) address space for erasing. It includes two elements to decode the first half of Large block select and the second half of Large block select.	Bit-mapped value such that the least significant bit is at bit 0 of Large block region (if available). Select the block in the Large address space to be erased by setting 1 to the appropriate bit of *nLargeBlockSelect*. If there is not any block to be erased in the Large address space, *nLargeBlockSelect* must be set to 0.

Return values

Table 11. Return values for FlashErase

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_ERASE_OPTION C55_ERROR_BUSY C55_ERROR_FACTORY_OP

Troubleshooting

Table 12. Troubleshooting for FlashErase

Error codes	Possible causes	Solution
C55_ERROR_ERASE_OPTION	Invalid erase option.	Use one of the valid values for the option.
C55_ERROR_BUSY	New erase operation cannot be performed because there is program/erase sequence in progress on the Flash module.	Wait until all previous program/erase operations on the Flash module finish. Possible cases that erase cannot start are: – erase in progress (MCR-ERS is high); – program in progress (MCR-PGM is high);
C55_ERROR_FACTORY_OP	The factory erase could not be performed.	Factory erase is locked by the system due to the data at the UTest NVM ‘diary’ location.

Comments

'FlashErase' always uses main interface to complete an erase operation and ignores the value of the ‘*mainInterfaceFlag'* in the SSD configuration structure. However, it is recommended that user should explicitly set this flag value to TRUE before calling *'FlashErase'*.

The *eraseOption* input argument provides an option for user to select his expected erase operation. If user wants to do factory erase, he must select *eraseOption* as C55_ERASE_MAIN_FERS or C55_ERASE_UTEST_FERS. If user wants to do normal erase operation on main array, *eraseOption* must be C55_ERASE_MAIN and lastly, user must select C55_ERASE_UTEST to do erase operation on UTest block.

The factory erase feature can be used to provide a faster erase. But the feature cannot be performed if the data at “diary” location in the UTest NVM space contains at least one zero at reset. In that case, each try to perform factory erase causes the error C55_ERROR_FACTORY_OP be returned.

The inputs *lowBlockSelect*, *midBlockSelect, highBlockSelect* and *nLargeBlockSelect* are bit-mapped arguments that are used to select the blocks to be erased in the Low/Mid/High/Large address spaces of main array. The selection of the blocks of the main array is determined by setting/clearing the corresponding bit in *lowBlockSelect*, *midBlockSelect, highBlockSelect* or *nLargeBlockSelect*.

The bit allocations for blocks in one address space are: the least significant bit is corresponding to 16K block region and start with block 0 (if available), then 32K block region (if available), then 64K block region (if available), and lastly 8K block region (if available). The following diagrams show the formats of *lowBlockSelect*, *midBlockSelect, highBlockSelect* and *nLargeBlockSelect* for the C55 module.

The Large block select includes two elements to decode the block selection for first 32 blocks (from bit 0 to bit 31) and second 32 blocks (from bit 32 to upper bits) separately.

Below is example for block allocation and bit map for specific Flash module with two blocks for each block size in low, middle or high address space. The invalid blocks are marked as reserved. And the number of valid bits may be various according to specific Flash module.

Table 13. Bit allocation for blocks in low address space

MSB							LSB
bit 31	**…**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
reserved	…	64K block 1	64K block 0	32K block 1	32K block 0	16K block 1	16K block 0

Table 14. Bit allocation for blocks in middle address space

MSB							LSB
bit 31	**…**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
reserved	…	64K block 1	64K block 0	32K block 1	32K block 0	16K block 1	16K block 0

Table 15. Bit allocation for blocks in high address space

MSB											LSB
bit 31	**…**	**bit 9**	**bit 8**	**bit 7**	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
reserved	…	8K block 3	8K block 2	8K block 1	8K block 0	64K block 1	64K block 0	32K block 1	32K block 0	16K block 1	16K block 0

Table 16. Bit Allocation for Blocks in the first Large Address Space

MSB							LSB
bit 31	**…**	**bit 16**	**bit 15**	**bit 14**	**…**	**bit 1**	**bit 0**
block 31	…	block 16	block 15	block 14	…	block 1	block 0

Table 17. Bit allocation for blocks in the second large address space

MSB							LSB
bit 31	**…**	**bit 16**	**bit 15**	**bit 14**	**…**	**bit 1**	**bit 0**
reserved	…	reserved	block 47	block 46	…	block 33	block 32

If the selected main array blocks or UTest block are locked for erasing, those blocks are not erased, but '*FlashErase'* still returns C55_OK. User needs to check the erasing result with the '*BlankCheck*' function.

It is impossible to erase any Flash block when a program or erase operation is already in progress on C55 module. '*FlashErase'* returns C55_ERROR_BUSY when trying to do so. In addition, when '*FlashErase'* is running, it is unsafe to read the data from the Flash partitions having one or more blocks being erased. Otherwise, it causes a Read-While-Write error.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.3 FlashEraseAlternate

Description

This function is to do erase operation for single block on single Flash module according to user's input arguments via alternate interface. The targeted Flash module status is checked in advance to return relevant error code if any. This function only set the high voltage without waiting for the operation to be finished. Instead, user must call '*FlashCheckStatus'* function to confirm the successful completion of this operation.

Prototype

```
UINT32 FlashEraseAlternate (PSSD_CONFIG pSSDConfig,
UINT32 interlockAddress);
```

Arguments

Table 18. Arguments for FlashEraseAlternate

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
interlockAddress	The interlock address which points to the block needs to be erased.	The *interlockAddress* must fall in the block that user wants to erase and must be aligned to word.

Return values

Table 19. Return values for FlashEraseAlternate

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_BUSY C55_ERROR_ALIGNMENT

Troubleshooting

Table 20. Troubleshooting for FlashEraseAlternate

Returned Error Bits	Description	Solution
C55_ERROR_BUSY	New erase operation cannot be performed because there is program/erase sequence in progress on the Flash module.	Wait until all previous program/erase operations on the Flash module to finish. Possible cases that erase cannot start are: – erase in progress (MCR-ERS is high); – program in progress (MCR-PGM is high);
C55_ERROR_ALIGNMENT	The input argument of *interlockAddress* is not aligned by word.	The input argument of *interlockAddress* must be aligned by word.

Comments

FlashEraseAlternate' always uses alternate interface to complete an erase operation and ignores the value of the '*mainInterfaceFlag'* in the SSD configuration structure. However, it is recommended that user should explicitly set this flag value to FALSE before calling *FlashEraseAlternate'*.

The '*FlashEraseAlternate'* must not be used to erase any block in the Large address space. In that case the function only returns C55_OK without doing the operation.

If the selected main array blocks are locked for erasing, those blocks are not erased, but '*FlashEraseAlternate'* still returns C55_OK. User needs to check the erasing result with the '*BlankCheck*' function.

It is impossible to erase any Flash block when a program or erase operation is already in progress on C55 module. '*FlashEraseAlternate'* returns C55_ERROR_BUSY when trying to do so.

In addition, when '*FlashEraseAlternate'* is running, it is unsafe to read the data from the Flash partitions having one or more blocks being erased. Otherwise, it causes a Read-While-Write error.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.4 BlankCheck

Description

This function is to do blank check for the previous erase operation. It verifies whether the expected Flash range is blank or not. In case of mismatch, the failed address and failed destination is saved and relevant error code is returned.

This function only does blank check for given number of bytes which can terminate this function within expected time interval. Thus, if user wants to do blank check for large size, the rest of information need to be blank checked is stored in "pCtxData" variable and '*FlashCheckStatus'* must be called periodically to do the next blank check for next destination based on all data provided in "pCtxData".

Prototype

```
UINT32 BlankCheck (PSSD_CONFIG pSSDConfig,
UINT32 dest,
UINT32 size,
UINT32 *pFailedAddress,
UINT32 *pFailedData,
PCONTEXT_DATA pCtxData);
```

Arguments

Table 21. Arguments for BlankCheck

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
dest	Destination address to be checked.	Any accessible address aligned on word boundary in either main array or UTest block.
size	Size, in bytes, of the Flash region to check.	If size = 0, the return value is C55_OK. It should be word aligned and its combination with *dest* should fall in either main array or UTest block.
pFailedAddress	Return the address of the first non-blank Flash location in the checking region	Only valid when this function returns C55_ERROR_NOT_BLANK.
pFailedData	Return the content of the first non-blank Flash location in the checking region.	Only valid when this function returns C55_ERROR_NOT_BLANK.
pCtxData	Address of context data structure.	A data structure for storing context variables.

Return values

Table 22. Return values for BlankCheck

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_ALIGNMENTC55_ERROR_NOT_BLANK

Troubleshooting

Table 23. Troubleshooting for BlankCheck

Returned Error Bits	Description	Solution
C55_ERROR_ALIGNMENT	The *dest, size* provided by user is not aligned by word.	The *dest, size* must be word aligned.
C55_ERROR_NOT_BLANK	There is a non-blank area within targeted Flash range.	Call '*FlashErase'* to re-erase the targeted Flash range and do blank check again.

Comments

If the blank checking fails, the first failing address is saved to *pFailedAddress*, and the failing data in Flash is saved to *pFailedData*. The contents pointed by *pFailedAddress* and

pFailedData are updated only when there is a non-blank location in the checked Flash range.

If user wants to do blank check for large size, this Flash size is divided into many small portions defined by NUM_WORDS_BLANK_CHECK_CYCLE such that blank check for one small portion can be finished within expected time interval. In this case, '*BlankCheck'* function plays a role to kick-off this blank check operation by finishing blank check for the first portion after back-up all necessary information to *pCtxData* variable. And blank check from the second portion is done within '*FlashCheckStatus'* function. Thus, user must call '*FlashCheckStatus'* to finish all his expected operations defined by *size* argument.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.5 FlashProgram

Description

This function is to do program operation for single or multi-programmable size via different interface on targeted Flash module according to user's input arguments. The targeted Flash module status is checked in advance to return relevant error code if any. This function only set the high voltage without waiting for the operation to be finished. Instead, user must call '*FlashCheckStatus'* function to confirm the successful completion of this operation.

In case of programming for multi-programmable size, the rest of information need to be programmed is stored in "pCtxData" variable and the '*FlashCheckStatus'* function is called periodically by user to confirm the successful completion of the previous destination and once finish, this function invokes '*FlashProgram*' more times to program the next destination based on data provided in "pCtxData" until finish all.

Prototype

```
UINT32 FlashProgram (PSSD_CONFIG pSSDConfig,
BOOL factoryPgmFlag,
UINT32 dest.
UINT32 size,
UINT32 source,
PCONTEXT_DATA pCtxData);
```

Arguments

Table 24. Arguments for FlashProgram

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
factoryPgmFlag	A flag indicate to do whether factory program or not.	TRUE to do factory program, FALSE to do normal program.
dest	Destination address to be programmed in Flash memory.	Any accessible address aligned on double word boundary in either main array or UTest space.

Table 24. Arguments for FlashProgram (continued)

Argument	Description	Range
size	Size, in bytes, of the Flash region to be programmed.	If size = 0, C55_OK is returned. It should be multiple of word and its combination with *dest* should fall in either main array or UTest block.
source	Source program buffer address.	This address must reside on word boundary.
pCtxData	Address of context data structure.	A data structure for storing context variables

Return values

Table 25. Return values for FlashProgram

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_ALTERNATE C55_ERROR_ALIGNMENTC55_ERROR_BUSY C55_ERROR_FACTORY_OP

Troubleshooting

Table 26. Troubleshooting for FlashProgram

Returned Error Bits	Description	Solution
C55_ERROR_ALTERNATE	This error occurs when user wants to perform factory program via the alternate interface.	Use main interface if want to perform factory program or perform normal program if want to use alternate interface.
C55_ERROR_ALIGNMENT	This error indicates that *dest*/*size*/*source* isn't properly aligned.	Check if *dest* is aligned on double word (64-bit) boundary. Check if *size* and *source* are aligned on word boundary.
C55_ERROR_BUSY	There is program operation is in progress or erase operation is going on and not in suspended state.	Wait for the on-going high voltage operation to finish. Flash program operation can be started if: – There is no program or erase operation being in progress. – If erase operation is in progress and it must be in suspended state.
C55_ERROR_FACTORY_OP	The factory program could not be performed due to the data at the 'diary' location in the UTest NVM contains at least one zero.	Check the data at the 'diary' location in the UTest NVM or just perform a normal program.

Comments

After performing a program, '*ProgramVerify*' should be used to verify the programmed data is correct or not.

'*FlashProgram*' checks the *mainInterfaceFlag* in the SSD configuration to decide which interface to be used for the operation, the main interface or the alternate interface. User should explicitly set this parameter before calling the function.

This function also provides a faster method for user to perform, factory program. But the feature cannot be performed if the data at "diary" location in the UTest NVM space contains at least one zero at reset. In that case, each try to perform factory program cause the error C55_ERROR_FACTORY_OP be returned.

If the selected main array blocks are locked for programming, those blocks are not programmed, and '*FlashProgram*' returns C55_OK.

If user wants to program to Large block space via alternate interface, this function still returns C55_OK without doing any program operation.

It is impossible to program any Flash block when a program or erase operation has already been in progress on C55 module. '*FlashProgram*' returns C55_ERROR_BUSY when doing so. However, user can use the '*FlashSuspend*' function to suspend an on-going erase operation on one block to perform a program operation on another block.

It is unsafe to read the data from the Flash partitions having one or more blocks being programmed when '*FlashProgram*' is running. Otherwise, it causes a Read-While-Write error.

If user wants to do program for multi-programmable size, this function plays a role to kick-off this operation by finishing program for the first programmable size after back-up all necessary information to *pCtxData* variable. And programming from the second programmable size is done within '*FlashCheckStatus'* function. Thus, user must call '*FlashCheckStatus'* to finish all his expected operations defined by *size* argument.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API. And Flash location must be in erased state before calling '*FlashProgram'* API.

2.7.6 ProgramVerify

Description

This function is to verify the previous program operation. It verifies if the programmed Flash range matches the corresponding source data buffer. In case of mismatch, the failed address, failed destination and failed source are saved and relevant error code are returned.

This function only does verification for given number of bytes which can terminate this function within expected time interval. Thus, if user wants to do Flash verification for large size, the rest of information need to be verified is stored in "pCtxData" variable and '*FlashCheckStatus'* must be called periodically to do the next verification for next destination based on all data provided in "pCtxData".

Prototype

```
UINT32 ProgramVerify (PSSD_CONFIG pSSDConfig,
UINT32 dest,
UINT32 size,
UINT32 source,
UINT32 *pFailedAddress,
UINT32 *pFailedData,
UINT32 *pFailedSource,
PCONTEXT_DATA pCtxData);
```

Arguments

Table 27. Arguments for ProgramVerify

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
dest	Destination address to be verified in Flash memory.	Any accessible address aligned on word boundary in main array or UTest block.
size	Size, in byte, of the Flash region to verify.	If size = 0, C55_OK is returned. It must be word aligned and its combination with *dest* should fall within main array or UTest block.
source	Verify source buffer address.	This address must reside on word boundary.
pFailedAddress	Return first failing address in Flash.	Only valid when the function returns C55_ERROR_VERIFY.
pFailedData	Returns first mismatch data in Flash.	Only valid when this function returns C55_ERROR_VERIFY.
pFailedSource	Returns first mismatch data in buffer.	Only valid when this function returns C55_ERROR_VERIFY.
pCtxData	Address of context data structure.	A data structure for storing context variables

Return values

Table 28. Return values for ProgramVerify

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_ALIGNMENTC55_ERROR_VERIFY

Troubleshooting

Table 29. Troubleshooting for ProgramVerify

Returned Error Bits	Description	Solution
C55_ERROR_ALIGNMENT	This error indicates that *dest*/*size*/*source* isn't properly aligned.	Check if *dest* , *size* and *source* are aligned on word (32-bit) boundary.
C55_ERROR_VERIFY	There is a mismatch between destination and source data.	Check if the data in source is correct. If yes, the previous program operation is failed. User should re-erase that Flash location and do program again.

Comments

The contents pointed by *pFailedAddress*, *pFailedData* and *pFailedSource* are updated only when there is a mismatch between the source and destination regions.

If user wants to do program verify for large size, this Flash size is divided into many small portions defined by NUM_WORDS_PROGRAM_VERIFY_CYCLE such that verification for one small portion can be finished within expected time interval. In this case, '*ProgramVerify'* function plays a role to kick-off this verification operation by finishing verification for the first portion after back-up all necessary information to *pCtxData* variable. And verification from the second portion is done within '*FlashCheckStatus'* function. Thus, user must call '*FlashCheckStatus'* to finish all his expected operations defined by *size* argument.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.7 CheckSum

Description

This function performs a 32-bit sum over the specified Flash memory range without carry, which provides a rapid method for data integrity checking.

This function only does Flash check sum for given number of bytes which can terminate this function within expected time interval. Thus, if user wants to do check sum for large size, the rest of information need to be checked sum is stored in "pCtxData" variable and '*FlashCheckStatus'* must be called periodically to do the next check sum for next destination based on all data provided in "pCtxData".

Prototype

```
UINT32 CheckSum (PSSD_CONFIG pSSDConfig,
UINT32 dest,
UINT32 size,
UINT32 *pSum,
PCONTEXT_DATA pCtxData);
```

Arguments

Table 30. Arguments for CheckSum

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
dest	Destination address to be summed in Flash memory.	Any accessible address aligned on word boundary in either main array or UTest block.
size	Size, in bytes, of the Flash region to check sum.	If size is 0 and the other parameters are all valid, C55_OK is returned. It must be word aligned and its combination with *dest* should fall within main array or UTest block.
pSum	Returns the sum value.	0x00000000 - 0xFFFFFFFF. Note that this value is only valid when the function returns C55_OK. User must not pass to this function with NULL pointer of *pSum*.
pCtxData	Address of context data structure.	A data structure for storing context variables.

Return values

Table 31. Return values for CheckSum

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_ALIGNMENT

Troubleshooting

Table 32. Troubleshooting for CheckSum

Returned Error Bits	Description	Solution
C55_ERROR_ALIGNMENT	This error indicates that *dest*/*size* isn't properly aligned.	Check if *dest* and *size* are aligned on word (32-bit) boundary.

Comments

In order to provide correct *pSum* calculation, this input argument must not be NULL pointer. However, this API does not return any error code if user tries doing so.

If user wants to do checksum for large size, this Flash size is divided into many small portions defined by NUM_WORDS_CHECK_SUM_CYCLE such that checksum for one small portion can be finished within expected time interval. In this case, '*CheckSum*' function plays a role to kick-off this operation by finishing checksum for the first portion after back-up all necessary information to *pCtxData* variable. And checksum from the second portion is done within '*FlashCheckStatus*' function. Thus, user must call '*FlashCheckStatus*' to finish all the expected operations defined by *size* argument.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.8 FlashCheckStatus

Description

This function checks the status of on-going high voltage operation in user mode or status of array integrity check in user test mode. The user's application code should call this function to determine whether the operation is done or failed or in progress. In addition, this function is used to recover the un-completed task in *FlashProgram, ProgramVerify, CheckSum , BlankCheck* in case user wants to call those functions with very big size.

In case of invoking program operation for multi-programmable size, after confirming that the previous program operation has been finished successfully, this function calls *FlashProgram* one more time to do the next program operation at next destination.

In case of invoking Flash verify operation for large size, this function calls *FlashVerify* one more time to do verification for the next portion of data.

In case of invoking blank check operation for large size, this function calls *BlankCheck* one more time to do blank check for the next portion of data.

In case of invoking check sum for large size, this function calls *CheckSum* one more time to do check sum for the next portion of data.

User must provide *modeOp* input argument with appropriate value to determine which operation needs to be checked by this function. Below list defines all possible cases to call this function:

Call FlashCheckStatus for program operation.

Call FlashCheckStatus for erase operation.

Call FlashCheckStatus for user's test mode.

Call FlashCheckStatus for Flash verification.

Call FlashCheckStatus for blank check.

Call FlashCheckStatus for check sum.

User must provide *pCtxData* input argument which is a pointer to the context data structure for each Flash function being checked for status. The context data structure contains a function pointer which must be manually set up for each Flash operation (program, blank check, program verify, check sum) to be checked for status. It is recommended to keep a separate context data structure for each type of Flash operation. As an example, please refer to the demo code included in the release package. Below is a code snippet.

```
CONTEXT_DATA dummyCtxData; // no context for erase and user test operation
CONTEXT_DATA pgmCtxData;
CONTEXT_DATA bcCtxData;
CONTEXT_DATA pvCtxData;
CONTEXT_DATA csCtxData;
/* set up function pointers in context data */
pgmCtxData.pReqCompletionFn = pFlashProgram;
bcCtxData.pReqCompletionFn = pBlankCheck;
```

```
pvCtxData.pReqCompletionFn = pProgramVerify;
csCtxData.pReqCompletionFn = pCheckSum;
```

Prototype

```
UINT32 FlashCheckStatus (PSSD_CONFIG pSSDConfig,
UINT8 modeOp,
UINT32 *opResult,
PCONTEXT_DATA pCtxData);
```

Arguments

Table 33. Arguments for FlashCheckStatus

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
modeOp	To specify the operation needs to be checked.	Must be one of the values: – C55_MODE_OP_PROGRAM – C55_MODE_OP_ERASE – C55_MODE_OP_PROGRAM_VERIFY – C55_MODE_OP_BLANK_CHECK – C55_MODE_OP_CHECK_SUM – C55_MODE_OP_USER_TEST_CHECK
opResult	To store result of the operation.	The values for this variable are depend on the operation being checked. For PROGRAM operation, they are: – C55_OK – C55_ERROR_PGOOD For ERASE operation, they are: – C55_OK – C55_ERROR_EGOOD For PROGRAM_VERIFY operation, they are: – C55_OK – C55_ERROR_VERIFY For BLANK_CHECK operation, they are: – C55_OK – C55_ERROR_NOT_BLANK For CHECK_SUM operation, it is always C55_OK. For USER_TEST_CHECK operation, they are: – C55_OK – C55_ERROR_MISMATCH
pCtxData	Address of a context data structure.	A data structure for storing context variables

Return values

Table 34. Return values for FlashCheckStatus

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_INPROGRESS C55_DONE C55_ERROR_MODE_OP All possible states in '*FlashSuspend()*' All possible states in '*FlashArrayIntegritySuspend()*'

Troubleshooting

Table 35. Troubleshooting for FlashCheckStatus

Returned Error Bits	Description	Solution
C55_ERROR_MODE_OP	User provides invalid *modeOp* argument.	The *modeOp* must be one of the values provided on Table 33.

Comments

User should call this function periodically until the whole operation finishes.

This function can also be called inside an interrupt procedure for program/erase to take the full advantage of interrupt. Each time the interrupt procedure is called, '*FlashCheckStatus*' gets called to continue to complete the whole operation.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit()*' API.

2.7.9 FlashSuspend

Description

This function checks if there is any high voltage operation being in progress on the C55 module and if this operation can be suspended. This function suspends the ongoing operation if it can be suspended.

Prototype

```
UINT32 FlashSuspend (PSSD_CONFIG pSSDConfig,
UINT8 *suspendState);
```

Arguments

Table 36. Arguments for FlashSuspend

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
suspendState	Indicate the suspend state of C55 module after the function being called.	All state values are enumerated in *Table 38*.

Return values

Table 37. Return values for FlashSuspend

Type	Description	Possible values
UINT32	Successful completion of this function.	C55_OK

Troubleshooting

None.

Comments

After calling this function, read is allowed on main array space without any Read-While-Write error. But data read from the blocks targeted for programming or erasing will be indeterminate even if the operation is suspended.

Following table defines and describes various suspend states and associated suspend codes.

Table 38. Suspend State Definitions

Argument	Code	Description	Valid operation after suspend
C55_SUS_NOTHING	10	There is no program/erase operation.	Erasing operation, programming operation and read are valid on main array space.
C55_PGM_WRITE	11	There is a program sequence in interlock write stage.	Only read is valid on main array space.
C55_ERS_WRITE	12	There is an erase sequence in interlock write stage.	Only read is valid on main array space.
C55_ERS_SUS_PGM_WRITE	13	There is an erase-suspend program sequence in interlock write stage.	Only read is valid on main array space.
C55_PGM_SUS	14	The program operation is in suspended state.	Only read is valid on main array space.

Table 38. Suspend State Definitions

Argument	Code	Description	Valid operation after suspend
C55_ERS_SUS	15	The erase operation on main array is in suspended state.	Programming/Read operation is valid on main array space.
C55_ERS_SUS_PGM_SUS	16	The erase-suspended program operation is in suspended state.	Only read is valid on main array space.

This function should be used together with '*FlashResume*'. If *suspendState* is C55_PGM_SUS or C55_ERS_SUS or C55_ERS_SUS_PGM_SUS, then '*FlashResume*' should be called in order to resume the operation.

The table below lists the Suspend State against to the Flash block status.

Table 39. Suspending State vs. C55 Status

suspendState	EHV	ERS	ESUS	PGM	PSUS
C55_SUS_NOTHING	X	0	X	0	X
C55_PGM_WRITE	0	0	X	1	0
C55_ERS_WRITE	0	1	0	0	X
C55_ESUS_PGM_WRITE	0	1	1	1	0
C55_PGM_SUS	1	0	X	1	0
	X	0	X	1	1
C55_ERS_SUS	1	1	0	0	X
	X	1	1	0	X
	X	1	1	0	X
C55_ERS_SUS_PGM_SUS	1	1	1	1	0
	X	1	1	1	1

The values of EHV, ERS, ESUS, PGM and PSUS represent the C55 status at the entry of '*FlashSuspend*'.

0: Logic zero; 1: Logic one; X: Do-not-care.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.10 FlashResume

Description

This function checks if there is any suspended erase or program operation on the C55 module, and resumes the suspended operation if there is any.

Prototype

```
UINT32 FlashResume (PSSD_CONFIG pSSDConfig,
UINT8* resumeState);
```

Arguments

Table 40. Arguments for FlashResume

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
resumeState	Indicate the resume state of C55 module after the function being called.	All state values are listed in *Table 42*

Return values

Table 41. Return values for FlashResume

Type	Description	Possible values
UINT32	Successful completion of this function.	C55_OK

Troubleshooting

None.

Comments

This function resumes one operation if there is any operation is suspended. For instance, if a program operation is in suspended state, it is resumed. If an erase operation is in suspended state, it is resumed too. If an erase-suspended program operation is in suspended state, the program operation is resumed prior to resuming the erase operation.

Following table defines and describes various resume states and associated resume codes.

Table 42. Resume state definitions

Code Name	Value	Description
C55_RES_NOTHING	20	No program/erase operation to be resumed
C55_RES_PGM	21	A program operation is resumed
C55_RES_ERS	22	A erase operation is resumed
C55_RES_ERS_PGM	23	A suspended erase-suspended program operation is resumed

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.11 GetLock

Description

This function checks the block locking status of Low/Middle/High/Large address spaces in the C55 module via either main or alternate interface.

Prototype

```
UINT32 GetLock (PSSD_CONFIG pSSDConfig,
UINT8 blkLockIndicator,
UINT32 *blkLockState);
```

Arguments

Table 43. Arguments for GetLock

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
blkLockIndicator	Indicating the address space which determines the address space block locking register to be checked.	Refer to *Table 46* for valid values for this parameter.
blkLockState	Returns the blocks' locking status in the given address space	Bit mapped value indicating the locking status of the specified address space. 1: The block is locked from program/erase. 0: The block is ready for program/erase

Return values

Table 44. Return values for GetLock

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_BLOCK_INDICATOR C55_ERROR_ALTERNATE

Troubleshooting

Table 45. Troubleshooting for GetLock

Returned Error Bits	Possible causes	Solution
C55_ERROR_BLOCK_INDICATOR	The input blkLockIndicator is invalid.	Set this argument to correct value listed in *Table 46*.
C55_ERROR_ALTERNATE	User calls this function to get lock status for Large block space via alternate interface.	Alternate interface does not support for Large block space.

Comments

Following table defines and describes various *blkLockIndicator* values.

Table 46. Lock indicator definitions

Code Name	Value	Description
C55_BLOCK_LOW	0	Block lock protection of low address space.
C55_BLOCK_MID	1	Block lock protection of mid address space.
C55_BLOCK_HIGH	2	Block lock protection of high address space.
C55_BLOCK_LARGE_FIRST	3	Block lock protection of the first Large address space (from block 0 to block 31).
C55_BLOCK_LARGE_SECOND	4	Block lock protection of the second Large address space (from block 32 to upper block numbering).
C55_BLOCK_UTEST	5	Block lock protection of the UTest block.

The output parameter *blkLockState* returns a bit-mapped value indicating the block lock status of the specified address space. A main array block is locked from program/erase if its corresponding bit is set.

The indicated address space determines the valid bits of *blkLockState*. For either Low/Mid/High/Large address spaces, if blocks corresponding to valid block lock state bits are not present (due to configuration or total memory size), values for these block lock state bits are always 1 because such blocks are locked by hardware on reset. These blocks cannot be unlocked by software with '*SetLock*' function.

If user uses the alternate interface to get the lock protection for the Large address space, the error code C55_ERROR_ALTERNATE is returned to indicate that the interface does not support this operation.

The bit allocations for blocks in one address space are: the least significant bit is corresponding to 16K block region and start with block 0 (if available), then 32K block region (if available) then lastly 64K block region (if available) and lastly 8K block region (if available).

The Large block space is divided into two separate sections corresponding two different block lock indicators. The C55_BLOCK_LARGE_FIRST lock indicator represents the first 32 blocks (from bit 0 to bit 31) of Large block space (128K/256K) and the C55_BLOCK_ LARGE_SECOND lock indicator represents the second 32 blocks (from bit 32 to upper bits) of Large block space (128K/256K).

Below is example for the formats of *blkLockState* in the C55 Flash module according to specific address space. In particular, this is an example with two blocks for each block size in low, middle or high address space and 48 blocks for Large block (128K/256K) address space. The invalid blocks are marked as reserved. And the number of valid bits may be various according to specific Flash module.

Table 47. *blkLockState* in low address space

MSB							LSB
bit 31	**…**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
reserved	…	64K block 1	64K block 0	32K block 1	32K block 0	16K block 1	16K block 0

Table 48. *blkLockState* in middle address space

MSB							LSB
bit 31	...	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
reserved	...	64K block 1	64K block 0	32K block 1	32K block 0	16K block 1	16K block 0

Table 49. *blkLockState* in high address space

MSB							LSB
bit 31	...	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
reserved	...	64K block 1	64K block 0	32K block 1	32K block 0	16K block 1	16K block 0

Table 50. *blkLockState* in the first large block (128K/256K) address space

MSB							LSB
bit 31	...	bit 16	bit 15	bit 14	...	bit 1	bit 0
block 31	...	block 16	block 15	block 14	...	block 1	block 0

Table 51. *blkLockState* in the second large block space (128K/256K) address space

MSB							LSB
bit 31	...	bit 16	bit 15	bit 14	...	bit 1	bit 0
reserved	...	reserved	block 47	block 46	...	block 33	block 32

Table 52. *blkLockState* in UTest block Space

MSB							LSB
bit 31	...	bit 16	bit 15	bit 14	...	bit 1	bit 0
reserved	...	reserved	reserved	reserved	...	reserved	block 0

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.12 SetLock

Description

This function will set the block lock state for Low/Middle/High/ Large (128K/256K) address space on the C55 module to protect them from program/erase via either main or alternate interface.

Prototype

```
UINT32 SetLock (PSSD_CONFIG pSSDConfig,
UINT8 blkLockIndicator,
```

```
UINT32 blkLockState);
```

Arguments

Table 53. Arguments for SetLock

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
blkLockIndicator	Indicating the address space and the protection level of the block lock register to be read.	Refer to *Table 46* for valid codes for this parameter.
blkLockState	The block locks to be set to the specified address space and protection level.	Bit mapped value indicating the lock status of the specified address space. 1: The block is locked from program/erase. 0: The block is ready for program/erase

Return values

Table 54. Return values for SetLock

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_BLOCK_INDICATOR C55_ERROR_ALTERNATE

Troubleshooting

Table 55. Troubleshooting for SetLock

Returned Error Bits	Possible causes	Solution
C55_ERROR_BLOCK_INDICATOR	The input blkLockIndicator is invalid.	Set this argument to correct value listed in *Table 46*.
C55_ERROR_ALTERNATE	User calls this function to set lock for Large block space via alternate interface.	Alternate interface does not support for Large block space.

Comments

See '*GetLock*' API.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.7.13 OverPgmProtGetStatus

Description

This function returns the over-program protection status via either main or alternate interface. This value shows blocks that are protected from being over programmed.

Prototype

```
UINT32 OverPgmProtGetStatus(PSSD_CONFIG pSSDConfig,
UINT8 blkProtIndicator,
UINT32 *blkProtState);
```

Arguments

Table 56. Arguments for OverPgmProtGetStatus

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
blkProtIndicator	The block indicator to get over-program protection status. This argument will determine which over-program protection register need to be accessed by this function.	The valid value for this argument is as same as that of *blkLockIndicator* argument in '*SetLock* 'function.
blkProtState	The bit map for over-program protection information of specific address space according to *blkProtIndicator* argument.	Bit-mapped value. 1: The block is protected from over-program. 0: The block is ready for over-program.

Return values

Table 57. Return values for OverPgmProtGetStatus

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_BLOCK_INDICATOR C55_ERROR_ALTERNATE

Troubleshooting

Table 58. Troubleshooting for OverPgmProtGetStatus

Returned Error Bits	Possible causes	Solution
C55_ERROR_BLOCK_INDICATOR	The input *blkProtIndicator* is invalid.	Set this argument to correct value listed in *Table 46*.
C55_ERROR_ALTERNATE	User calls this function to get over-program protection status via alternate interface.	Alternate interface does not support this operation.

Comments

If user uses the alternate interface to get the over program protection status for the Large address space, the error code C55_ERROR_ALTERNATE is returned to indicate that the interface does not support this operation.

The *blkProtState* is bit map allocation and it has the same definition with *blkLockState* of '*GetLock'* function. See '*GetLock'* function for more details.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.8 User Test Mode Functions

2.8.1 FlashArrayIntegrityCheck

Description

This function checks the array integrity of the Flash via main interface. The user specified address sequence is used for array integrity reads and the operation is done on the specified blocks. The MISR values calculated by the hardware is compared to the values passed by the user, if they are not the same, then an error code is returned.

In order to support asynchronous design, this function stores the necessary information to "pCtxData" (ex: user provided MISR value) and is terminated without waiting for completion of this operation. User should call '*FlashCheckStatus'* to check the on-going status of this function. And once finish, it will do comparison between MISR values provided by user which is currently stored in "pCtxData" and MISR values generated by hardware and return an appropriate code according to this compared result.

Prototype

```
UINT32 FlashArrayIntegrityCheck(PSSD_CONFIG pSSDConfig,
UINT32 lowEnabledBlocks,
UINT32 midEnabledBlocks,
UINT32 highEnabledBlocks,
NLARGE_BLOCK_SEL nLargeEnabledBlocks,
UINT8 breakOption,
UINT8 addrSeq,
PMISR pMisrValue,
PCONTEXT_DATA pCtxData);
```

Arguments

Table 59. Arguments for FlashArrayIntegrityCheck

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
lowEnabledBlocks	To select the array blocks in low address space for checking.	Refer to '*FlashErase*' for details.
midEnabledBlocks	To select the array blocks in mid address space for checking.	Refer to '*FlashErase*' for details.
highEnabledBlocks	To select the array blocks in high address space for checking.	Refer to '*FlashErase*' for details.
nLargeEnabledBlocks	To select the array blocks in Large block space (128K/256K) address space for checking.	Refer to '*FlashErase*' for details.
breakOption	To specify an option to allow stopping the operation on errors.	Must be one of the values: – C55_BREAK_NONE – C55 BREAK_ON_DBD (stop the operation on Double Bit Detection) – C55_BREAK_ON_DBD_SBC (stop the operation on Double Bit Detection or Single Bit Correction)
addrSeq	To determine the address sequence to be used during array integrity checks.	Must be one of below values: – C55_ADDR_SEQ_PROPRIETARY: this is meant to replicate sequences normal "user" code follows, and thoroughly check the read propagation paths. This sequence is proprietary – C55_ADDR_SEQ_LINEAR: this is just logically sequential. It should be noted that the time to run a sequential sequence is significantly shorter than the time to run the proprietary sequence.
pMISRValue	Address of a MISR structure contains the MISR values calculated by offline tool.	The individual MISR words can range from 0x00000000 - 0xFFFFFFFF
pCtxData	Address of a context data structure.	A data structure for storing context variables

Return values

Table 60. Return values for FlashArrayIntegrityCheck

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_ADDR_SEQ C55_ERROR_NO_BLOCK C55_ERROR_MISMATCH C55_ERROR_ALTERNATE

Troubleshooting

The trouble shooting given here comprises of hardware errors and input parameter error.

Table 61. Troubleshooting for FlashArrayIntegrityCheck

Returned Error Bits	Possible causes	Solution
C55_ERROR_MISMATCH	The MISR values calculated by the user are incorrect.	Re-calculate the MISR values using the correct Data and address sequence.
	The MISR values calculated by the Hardware are incorrect.	Hardware Error.
C55_ERROR_NO_BLOCK	None of the Blocks are enabled for Array Integrity Check	Enable any of the blocks using variables *lowEnabledBlocks, midEnabledBlocks, highEnabledBlocks* or *nLargeEnabledBlocks.*
C55_ERROR_ADDR_SEQ	User provides invalid address sequence input argument.	The address sequence input argument must be either proprietary (C55_ADDR_SEQ_PROPRIETARY) or sequential (C55_ADDR_SEQ_LINEAR). Any other value is unacceptable.
C55_ERROR_ALTERNATE	User calls this function via alternate interface.	Alternate interface does not support this operation.

Comments

The inputs *lowEnabledBlocks*, *midEnabledBlocks*, *highEnabledBlocks* and *nLargeEnabledBlock* are bit-mapped arguments that are used to select the blocks to be evaluated in the Low/Mid/High/ Large block (128K/256K) address spaces of main array. The selection of the blocks of the main array is determined by setting/clearing the corresponding bit in *lowEnabledBlocks*, *midEnabledBlocks, highEnabledBlocks* or *nLargeEnabledBlocks*.

For diagrams of block bit-map definitions of *lowEnabledBlocks*, *midEnabledBlocks*, *highEnabledBlocks* and *nLargeEnabledBlock*, refer to '*FlashErase'* function for more details.

In case user specifies a break option other than C55_BREAK_NONE, the function is stopped immediately if any Double Bit Detection or Single Bit Correction occurs. It is possible to resume the operation by calling *'FlashArrayIntegrityResume'* or start a new array integrity check.

If no blocks are enabled the C55_ERROR_NO_BLOCK error code is returned.

If user calls this function via alternate interface, the C55_ERROR_ALTERNATE error code is returned.

This function does not support to do array integrity check on UTest block.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.8.2 FlashArrayIntegritySuspend

Description

This function will check if there is an on-going array integrity check of the Flash and suspend it via main interface.

Prototype

```
UINT32 FlashArrayIntegritySuspend (PSSD_CONFIG pSSDConfig,
UINT8 *suspendState);
```

Arguments

Table 62. Arguments for FlashArrayIntegritySuspend

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
suspendState	Indicate the suspend state on user test mode after calling the function.	All state values are enumerated in *Table 65*.

Return values

Table 63. Return values for FlashArrayIntegritySuspend

Type	Description	Possible values
UINT32	Successful completion error code.	C55_OK C55_ERROR_ALTERNATE

Troubleshooting

Table 64. Troubleshooting for FlashArrayIntegritySuspend

Returned Error Bits	Possible causes	Solution
C55_ERROR_ALTERNATE	User calls this function via alternate interface.	Alternate interface does not support this operation.

Comments

If user calls this function via alternate interface, a return code of C55_ERROR_ALTERNATE is returned without doing any operation.

Following table defines and describes various suspend states and associated suspend codes.

Table 65. Suspend State Definitions

Argument	Code	Description
C55_SUS_NOTHING	10	There is no array integrity check/margin read operation in-progress.
C55_USER_TEST_SUS	17	The user test operation is in suspended state.

This function should be used together with '*FlashArrayIntegrityResume*'. If *suspendState* is C55_UTEST_SUS, then '*FlashArrayIntegrityResume*' should be called in order to resume the operation.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.8.3 FlashArrayIntegrityResume

Description

This function checks if there is an on-going array integrity check of the Flash being suspended and resume it via main interface.

Prototype

```
UINT32 FlashArrayIntegrityResume (PSSD_CONFIG pSSDConfig,
UINT8 *resumeState);
```

Arguments

Table 66. Arguments for FlashArrayIntegrityResume

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
resumeState	Indicate the resume state on user's test mode after calling the function.	All state values are enumerated in *Table 69*.

Return values

Table 67. Return values for FlashArrayIntegrityResume

Type	Description	Possible values
UINT32	Successful completion or error code.	C55_OK C55_ERROR_ALTERNATE

Troubleshooting

Table 68. Troubleshooting for FlashArrayIntegrityResume

Returned Error Bits	Possible causes	Solution
C55_ERROR_ALTERNATE	User calls this function via alternate interface.	Alternate interface does not support this operation.

Comments

If user calls this function via alternate interface, a return code of C55_ERROR_ALTERNATE is returned without doing any operation.

This function can also be used to resume an array integrity check/margin read check when it is stopped by a Double Bit Detection or a Single Bit Correction.

Following table defines and describes various resume states and associated resume codes.

Table 69. Resume state definitions

Argument	Code	Description
C55_RES_NOTHING	20	There is no array integrity check/margin read operation suspended.
C55_RES_USER_TEST	24	The user test operation is in in-progress state.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

2.8.4 UserMarginReadCheck

Description

This function checks the user margin reads of the Flash via main interface. The user specified margin level is used for reads and the operation is done on the specified blocks. The MISR values calculated by the hardware are compared to the values passed by the user, if they are not the same, then an error code is returned.

In order to support asynchronous design, this function stores the necessary information to "pCtxData" (ex: user provided MISR value) and is terminated without waiting for completion of this operation. User should call '*FlashCheckStatus'* to check the on-going status of this function. And once finish, it does comparison between MISR values provided by user which are currently stored in "pCtxData" and MISR values generated by hardware and return an appropriate code according to this compared result.

Prototype

```
UINT32 UserMarginReadCheck (PSSD_CONFIG pSSDConfig,
UINT32 lowEnabledBlocks,
UINT32 midEnabledBlocks,
UINT32 highEnabledBlocks,
NLARGE_BLOCK_SEL nLargeEnabledBlocks,
UINT8 breakOption,
UINT8 marginLevel,
PMISR pMisrValue,
PCONTEXT_DATA pCtxData);
```

Arguments

Table 70. Arguments for UserMarginReadCheck

Argument	Description	Range
pSSDConfig	Pointer to the SSD Configuration Structure.	The values in this structure are chip-dependent. Please refer to *Section 2.3* for more details.
lowEnabledBlocks	To select the array blocks in low address space for checking.	Refer to '*FlashErase*' for details.
midEnabledBlocks	To select the array blocks in mid address space for being evaluated.	Refer to '*FlashErase*' for details.
highEnabledBlocks	To select the array blocks in high address space for being evaluated.	Refer to '*FlashErase*' for details.
nLargeEnabledBlocks	To select the array blocks in Large (128K/256K) address space for being evaluated.	Refer to '*FlashErase*' for details.
breakOption	To specify an option to allow stopping the operation on errors.	Refer to '*FlashArrayIntegrityCheck*' for details.
marginLevel	To determine the margin level to be used during margin read checks.	Selects the margin level that is being checked. Must be one of the values: – C55_MARGIN_LEVEL_ERASE – C55_MARGIN_LEVEL_PROGRAM
pMISRValue	Address of a MISR structure contains the MISR values calculated by the user.	Refer to '*FlashArrayIntegrityCheck*' for details.
pCtxData	Address of a context data structure.	A data structure for storing context variables

Return values

Table 71. Return values for UserMarginReadCheck

Type	Description	Possible values
UINT32	Successful completion or error value.	C55_OK C55_ERROR_ALTERNATE C55_ERROR_MARGIN_LEVEL C55_ERROR_NO_BLOCK C55_ERROR_MISMATCH

Troubleshooting

Table 72. Troubleshooting for UserMarginReadCheck

Returned Error Bits	Possible causes	Solution
C55_ERROR_MISMATCH	The MISR values calculated by the user are incorrect.	Re-calculate the MISR values using the correct Data and margin level.
	The MISR values calculated by the Hardware are incorrect.	Hardware Error.
C55_ERROR_NO_BLOCK	None of the Blocks are enabled for Factory Margin Read Check	Enable any of the blocks using variables *lowEnabledBlocks, midEnabledBlocks, highEnabledBlocks and nLargeEnabledBlocks*
C55_ERROR_MARGIN_LEVEL	User provides invalid margin level.	The margin level input argument must be either program level (C55_MARGIN_LEVEL_PROGRAM) or erase level (C55_MARGIN_LEVEL_ERASE). Any other value is unacceptable.
C55_ERROR_ALTERNATE	User calls this function via alternate interface.	Alternate interface does not support this operation.

Comments

Refer to '*FlashArrayIntegrityCheck'* for details.

Assumptions

It assumes that the Flash block is initialized using a '*FlashInit*' API.

Appendix A Code sizes and stack usage

Table 73. Code size and stack usage for SPC574Kxx

API name	Code size (in bytes)	Stack usage (in bytes)
FlashInit()	192	48
FlashProgram()	312	96
ProgramVerify()	184	80
FlashErase()	440	80
FlashEraseAlternate()	110	N/A
FlashCheckStatus()	858	80
BlankCheck ()	154	64
CheckSum()	160	64
FlashSuspend()	240	48
FlashResume()	162	64
GetLock()	322	96
SetLock()	326	80
OverPgmProtGetStatus()	282	80
FlashArrayIntegrityCheck()	598	112
FlashArrayIntegrityResume()	182	64
FlashArrayIntegritySuspend()	126	48
UserMarginReadCheck()	620	112

Note: Code size is measured on Diab compiler with version 5.7.0.0 on vle mode and SPC574Kxx is selected.

Stack size is measured on CodeWarrior compiler v2.7 on SPC574Kxx.

Appendix B Write/erase times

Table 74. Write/erase times for SPC57EM80xx

Operation	Time (ms)
FlashProgram (PROGRAMMABLE_SIZE = 128)	0.128444444
ProgramVerify (NUM_WORDS_PROGRAM_VERIFY_CYCLE = 80)	0.282888889
CheckSum (NUM_WORDS_CHECK_SUM_CYCLE = 120)	0.282444444
FlashErase (one block)	0.031481481
BlankCheck (NUM_WORDS_BLANK_CHECK_CYCLE = 90)	0.314962963

Note: The timing values are measured on SPC57EM80xx device with 13.5MHz of system clock and on VLE mode.

Table 75. Write/erase times for SPC574Kxx

Operation	Time (ms)
FlashProgram (PROGRAMMABLE_SIZE = 128)	0.0192
ProgramVerify (NUM_WORDS_PROGRAM_VERIFY_CYCLE = 80)	0.038225
CheckSum (NUM_WORDS_CHECK_SUM_CYCLE = 120)	0.041325
FlashErase (one block)	0.00475
BlankCheck (NUM_WORDS_BLANK_CHECK_CYCLE = 90)	0.042525

Note: The timing values are measured on SPC574Kxx device with 80MHz of system clock and on VLE mode.

Appendix C System requirements

The C55 SSD is designed to support a single C55 Flash module embedded on microcontrollers. Before using this SSD on a different derivative microcontroller, user has to provide the information specific to the derivative through a configuration. The table below provides the hardware/tool which is necessary for using this driver.

Table 76. System requirements

Tool Name	Description	Version No
CodeWarrior IDE	Development tool	2.7
Diab PowerPC compiler	Compiler	5.7.0.0
GreenHills	Development tool	6.1.4
P/E	Debugger	

Appendix D Acronyms

Table 77. Acronyms

Abbreviation	Complete name
API	Application Programming Interface
BIU	Bus Interface Unit
ECC	Error Correction Code
EVB	Evaluation Board
RWW	Read While Write
SSD	Standard Software Driver

Appendix E Document references

1. SPC57EM80xx - 32-bit Power Architecture® based MCU with up to 4 Mbyte Flash and 304 Kbyte RAM memories (RM0314, DocID 022530)
2. SPC574Kxx - 32-bit Power Architecture® based MCU for automotive applications (RM0334, DocID 023671)

Revision history

Table 78. Document revision history

Date	Revision	Changes
11-Mar-2013	1	Initial release.