life.augmented

UM1635
User manual

Standard Software Driver for C90FL2 Flash module embedded on SPC56 L line microcontroller

Introduction

This document is the user manual for the Standard Software Driver (SSD) for single C90FL2 Flash module.

The SSD is a set of API's that enables user application to operate on the Flash module embedded on a microcontroller. The C90FL2 SSD contains a set of functions to program/erase a single C90FL2 Flash module.

The C90FL2 Standard Software Driver (SSD) provides the following API's:

- FlashInit
- FlashErase
- BlankCheck
- FlashProgram
- ProgramVerify
- CheckSum
- FlashSuspend
- FlashResume
- GetLock
- SetLock
- FlashDepletionRecover
- FlashArrayIntegrityCheck
- FlashECCLogicCheck
- FactoryMarginReadCheck

Contents

List of tables

1 Introduction

1.1 Document overview

This document is the user manual for the Standard Software Driver (SSD) for single C90FL2 Flash module. The roadmap for the document is as follows.

Section 1.2 shows the features of the driver.

Section 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 and Configuration Macros respectively. *Section 2.4* and *Section 2.5* describe the CallBack notifications, and return codes used for the driver. *Section 2.6* and *Section 2.7* provide the detailed description of normal mode and special mode standard software Flash Driver APIs' respectively.

Appendix A: CallBack timings provides the performance indexes. *Appendix B: System requirements* details the system requirement for the driver development. *Appendix C: Acronyms* lists the acronyms used. *Appendix D: Documentation References* and lists the documents referred and terms used in making of this document.

1.2 Features

The C90FL2 SSD provides the following features:

- Two sets of driver binaries built on Power Architecture instruction set technology and Variable-Length-Encoding (VLE) instruction set.
- Drivers released in binary c-array format to provide compiler-independent support for non-debug-mode embedded applications.
- Drivers 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 page-wise programming for fast programming.
- Position-independent and ROM-able
- Ready-to-use demos illustrating the usage of the driver
- Concurrency support via callback

2 API specification

2.1 General overview

The C90FL2 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 3 User Test APIs for checking the Array Integrity and the ECC Logic.

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
INT64	64-bits	signed long long
VINT64	64-bits	volatile signed long long
UINT64	64-bits	unsigned long long
VUINT64	64-bits	volatile unsigned long long

2.3 Configuration parameters and macros

The configuration parameter which is used for SSD operations is explained in this section. The configuration parameters are handled as structure. The user should correctly initialize the fields including *c90flRegBase*, *mainArrayBase*, *shadowRowBase*, *shadowRowSize*, *pageSize* and *BDMEnable* before passing the structure to SSD functions. The pointer to *CallBack* has to be initialized either to a null pointer or a valid *CallBack* function pointer.

Table 2. SSD configuration structure field definition

Parameter name	Type	Parameter description
c90flRegBase	UINT32	The base address of C90FL2 and BIU control registers.
mainArrayBase	UINT32	The base address of Flash main array.
mainArraySize	UINT32	The size of Flash main array in byte.
shadowRowBase	UINT32	The base address of shadow row
shadowRowSize	UINT32	The size of shadow row in byte.
lowBlockNum	UINT32	Block number of the low address space.
midBlockNum	UINT32	Block number of the mid address space.
highBlockNum	UINT32	Block number of the high address space.
pageSize	UINT32	The page size of the C90FL2 Flash
BDMEnable	UINT32	Defines the state of background debug mode (enable /disable)

The type definition for the structure is given below.

```
typedef struct _ssd_config
{
UINT32 c90flRegBase;
UINT32 mainArrayBase;
UINT32 mainArraySize;
UINT32 shadowRowBase;
UINT32 shadowRowSize;
UINT32 lowBlockNum;
UINT32 midBlockNum;
UINT32 highBlockNum;
UINT32 pageSize;
UINT32 BDMEnable;
} SSD_CONFIG, *PSSD_CONFIG;
```

Note: The macro value COMPILER_SELECT should be set to

CODE_WARRIOR – if CodeWarrior compiler is used for compiling

DIAB_COMPILER – if Diab compiler is used for compiling

2.4 Callback notification

The Standard Software Driver facilitates the user to supply a pointer to '*CallBack()*' function so that time-critical events can be serviced during C90FL2 Standard Software driver operations. Servicing watchdog timers is one such time critical event. If it is not necessary to provide the CallBack service, the user is able to disable it by a NULL function macro.

```
#define NULL_CALLBACK    ((void *) 0xFFFFFFFF)
```

The job processing callback notifications shall have no parameters and no return value.

2.5 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 3. Return codes

Name	Value	Description
C90FL_OK	0x00000000	The requested operation is successful.
C90FL_INFO_RWE	0x00000001	RWE bit is set before Flash operations.
C90FL_INFO_EER	0x00000002	EER bit is set before Flash operations.
C90FL_ERROR_ALIGNMENT	0x00000100	Alignment error.
C90FL_ERROR_RANGE	0x00000200	Address range error.
C90FL_ERROR_BUSY	0x00000300	New program/erase cannot be preformed while a high voltage operation is already in progress.
C90FL_ERROR_PGOOD	0x00000400	The program operation is unsuccessful.
C90FL_ERROR_EGOOD	0x00000500	The erase operation is unsuccessful.
C90FL_ERROR_NOT_BLANK	0x00000600	There is a non-blank Flash memory location within the checked Flash memory region.
C90FL_ERROR_VERIFY	0x00000700	There is a mismatch between the source data and the content in the checked Flash memory.
C90FL_ERROR_LOCK_INDICATOR	0x00000800	Invalid block lock indicator.
C90FL_ERROR_RWE	0x00000900	Read-while-write error occurred in previous reads.
C90FL_ERROR_PASSWORD	0x00000A00	The password provided cannot unlock the block lock register for register writes
C90FL_ERROR_AIC_MISMATCH	0x00000B00	In '*FlashArrayIntegrityCheck()*' the MISR values generated by the hardware do not match the values passed by the user.
C90FL_ERROR_AIC_NO_BLOCK	0x00000C00	In '*FlashArrayIntegrityCheck()*' no blocks have been enabled for Array Integrity check
C90FL_ERROR_FMR_MISMATCH	0x00000D00	In '*FactoryMarginReadCheck()*' the MISR values generated by the hardware do not match the values passed by the user.
C90FL_ERROR_FMR_NO_BLOCK	0x00000E00	In '*FactoryMarginReadCheck()*' no blocks have been enabled for Array Integrity check
C90FL_ERROR_ECC_LOGIC	0x00000F00	In '*FlashECCLogicCheck()*' the simulated ECC error has not occurred.

2.6 Normal mode functions

2.6.1 FlashInit()

Description

This function reads the Flash configuration information from the Flash control registers and initialize parameters in SSD configuration structure. ‘*FlashInit()*’ must be called prior to any other Flash operations.

Prototype

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

Arguments

Table 4. 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 5. Return values for FlashInit()

Type	Description	Possible values
UINT32	Indicates Flash initialization operation is successful.	C90FL_OK

Troubleshooting

None.

Comments

‘*FlashInit()*’ checks the C90FL_MCR_RWE and C90FL_MCR_EER bit, and clear them when any of them is set. If RWE bit is set, Flash program/erase operations can still be performed.

Assumptions

The user must correctly initialize the fields including *c90flRegBase*, *mainArrayBase*, *shadowRowBase*, *shadowRowSize*, *pageSize* and *BDMEnable* before passing the structure to the *FlashInit()* functions.

2.6.2 FlashErase()

Description

This function erases the enabled blocks in the main array or the shadow row. The relevant Flash module status is checked, and relevant error code is returned if there is any error.

Prototype

```
UINT32 FlashErase (PSSD_CONFIG pSSDConfig,
   BOOL shadowFlag,
   UINT32 lowEnabledBlocks,
   UINT32 midEnabledBlocks,
   UINT32 highEnabledBlocks,
   void (*CallBack)(void));
```

Arguments

Table 6. 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.
shadowFlag	Indicate either the main array or the shadow row to be erased.	TRUE: the shadow row is erased. The *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* is ignored; FALSE: The main array is erased. Which blocks are erased in low, mid and high address spaces are specified by *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* respectively.
lowEnabledBlocks	To select the array blocks in low address space for erasing.	Bit-mapped value. Select the block in the low address space to be erased by setting 1 to the appropriate bit of *lowEnabledBlocks*. If there is not any block to be erased in the low address space, *lowEnabledBlocks* must be set to 0.
midEnabledBlocks	To select the array blocks in mid address space for erasing.	Bit-mapped value. Select the block in the middle address space to be erased by setting 1 to the appropriate bit of *midEnabledBlocks*. If there is not any block to be erased in the middle address space, *midEnabledBlocks* must be set to 0.
highEnabledBlocks	To select the array blocks in high address space for erasing.	Bit-mapped value. Select the block in the high address space to be erased by setting 1 to the appropriate bit of *highEnabledBlocks*. If there is not any block to be erased in the high address space, *highEnabledBlocks* must be set to 0.
CallBack	Address of void call back function pointer.	Any addressable void function address. To disable it, use NULL_CALLBACK macro.

Return values

Table 7. Return values for FlashErase()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_BUSY C90FL_ERROR_EGOOD

Troubleshooting

Table 8. Troubleshooting for FlashErase()

Error Codes	Possible Causes	Solution

C90FL_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 (FLASH_MCR-ERS is high);
		– program in progress (FLASH_MCR-PGM is high);

C90FL_ERROR_ EGOOD	Erase operation failed.	Check if the C90FL2 is available and high voltage is applied to C90FL2. Then try to do the erase operation again.

Comments

When *shadowFlag* is set to FALSE, the *'FlashErase()'* function erases the blocks in the main array. It is capable of erasing any combination of blocks in the low, mid and high address spaces in one operation. If *shadowFlag* is TRUE, this function erases the shadow row.

The inputs *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* are bit-mapped arguments that are used to select the blocks to be erased in the Low/Mid/High 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 or *highEnabledBlocks*.

The bit allocations for blocks in one address space are: bit 0 is assigned to block 0, bit 1 to block 1, etc. The following diagrams show the formats of *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* for the C90FL2 module.

For low address space valid bits are from bit 0 to bit (*lowBlockNum – 1*). In which, *lowBlockNum* is the number of low blocks returned from FlashInit();

For middle address space valid bits are from bit 0 and bit (*midBlockNum – 1*). In which, *midBlockNum* is the number of middle blocks returned from FlashInit();

For high address space valid bits are from bit 0 to bit (*highBlockNum – 1*). In which, *highBlockNum* is the number of high blocks returned from FlashInit();

For example, below are bit allocations for blocks in Low/Mid/High Address Space of SPC56ELx:

Bit Allocation for Blocks in Low Address Space:

Table 9. Bit allocation for blocks in low address space

MSB **LSB**

bit 31	…	bit 10	bit 9	bit 8	…	bit 1	bit 0
reserved	…	reserved	block 9	block 8	…	block 1	block 0

Table 10. Bit allocation for blocks in middle address space

MSB LSB

bit 31	…	bit 4	bit 3	bit 2	bit 1	bit 0
reserved	…	reserved	reserved	reserved	block 1	block 0

Table 11. Bit allocation for blocks in high address space

MSB LSB

bit 31	…	bit 6	bit 5	bit 4	…	bit 1	bit 0
reserved	…	reserved	block 5	block 4	…	Block 1	Block 0

If the selected main array blocks or the shadow row is locked for erasing, those blocks or the shadow row is not erased, but '*FlashErase()*' still returns C90FL_OK. User needs to check the erasing result with the '*BlankCheck()*' function.

It is impossible to erase any Flash block or shadow row when a program or erase operation is already in progress on C90FL2 module. '*FlashErase()*' returns C90FL_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 the '*FlashInit()*' API. User provides the correct ssdConfig parameters to *FlashErase()* as returned by *FlashInit().*

2.6.3 BlankCheck()

Description

This function checks on the specified Flash range in the main array or shadow row for blank state. If the blank checking fails, the first failing address and the failing data in Flash block is saved.

Prototype

```
UINT32 BlankCheck (PSSD_CONFIG pSSDConfig,
   UINT32 dest,
   UINT32 size,
   UINT32 * pFailAddress,
   UINT64 *pFailData,
   void (*CallBack) (void ));
```

Arguments

Table 12. 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 double word boundary in main array or shadow row
size	Size, in bytes, of the Flash region to check.	If size = 0, the return value is C90FL_OK. It should be multiple of 8 and its combination with *dest* should fall in either main array or shadow row.
pFailAddress	Return the address of the first non-blank Flash location in the checking region	Only valid when this function returns C90FL_ERROR_NOT_BLANK.
pFailData	Return the content of the first non-blank Flash location in the checking region.	Only valid when this function returns C90FL_ERROR_NOT_BLANK.
CallBack	Address of void callback function	Any addressable void function address. To disable it, use NULL_CALLBACK macro.

Return values

Table 13. Return values for BlankCheck()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_ALIGNMENT C90FL_ERROR_RANGE C90FL_ERROR_NOT_BLANK

Troubleshooting

Table 14. Troubleshooting for BlankCheck()

Returned Error Bits	Description	Solution
C90FL_ERROR_ALIGNMENT	The *dest*/*size* are not properly aligned.	Check if *dest* and *size* are aligned on double word (64-bit) boundary.
C90FL_ERROR_RANGE	The area specified by *dest* and *size* is out of the valid C90FL2 array ranges.	Check *dest* and *dest*+*size*. The area to be checked must be within main array space or shadow space.
C90FL_ERROR_NOT_BLANK	There is a non-blank double word within the area to be checked.	Erase the relevant blocks and check again.

Comments

If the blank checking fails, the first failing address is saved to **pFailAddress* and the failing data in Flash is saved to **pFailData*. The contents pointed by *pFailAddress* and *pFailData* are updated only when there is a non-blank location in the checked Flash range.

Assumptions

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

2.6.4 FlashProgram()

Description

This function programs the specified Flash areas with the provided source data. Input arguments together with relevant Flash module status is checked and relevant error code is returned if there is any error.

Prototype

```
UINT32 FlashProgram (PSSD_CONFIG pSSDConfig,
   UINT32 dest,
   UINT32 size,
   UINT32 source,
   void (*CallBack)(void));
```

Arguments

Table 15. 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.
dest	Destination address to be programmed in Flash memory.	Any accessible address aligned on double word boundary in main array or shadow row.
size	Size, in bytes, of the Flash region to be programmed.	If size = 0, C90FL_OK is returned. It should be multiple of 8 and its combination with *dest* should fall in either main array or shadow row.
source	Source program buffer address.	This address must reside on word boundary.
CallBack	Address of void call back function pointer.	Any addressable void function address. To disable it use NULL_CALLBACK macro.

Return values

Table 16. Return values for FlashProgram()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_BUSY C90FL_ERROR_ALIGNMENT C90FL_ERROR_RANGE C90FL_ERROR_PGOOD

Troubleshooting

Table 17. Troubleshooting for FlashProgram()

Returned error bits	Description	Solution
C90FL_ERROR_BUSY	New program operation cannot be performed because the Flash module is busy with some operation and cannot meet the condition for starting a program operation.	Wait until the current operations finish. Conditions that program cannot start are: 1. program in progress (MCR-PGM high); 2. program not in progress (MCR-PGM low), but: – erase on main array in progress but not suspended; – erase on shadow row is in progress but not suspended;
C90FL_ERROR_ALIGNMENT	This error indicates that *dest*/*size*/*source* isn't properly aligned	Check if *dest* and *size* are aligned on double word (64-bit) boundary. Check if source is aligned on word boundary.
C90FL_ERROR_RANGE	The area specified by *dest* and *size* is out of the valid C90FL2 address range.	Check *dest* and *dest+size*. Both should fall in the same C90FL2 address ranges, i.e. both in main array or both in shadow row
C90FL_ERROR_PGOOD	Program operation failed because this operation cannot pass PEG check.	Repeat the program operation. Check if the C90FL2 is invalid or high voltage applied to C90FL2 is unsuitable.

Comments

If the selected main array blocks or the shadow row is locked for programming, those blocks or the shadow row are not programmed and '*FlashProgram()*' still returns C90FL_OK. User needs to verify the programmed data with '*ProgramVerify()*' function.

It is impossible to program any Flash block or shadow row when a program or erase operation is already in progress on C90FL2 module. '*FlashProgram()*' returns C90FL_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. The user has begun an erase operation on the main array or shadow row, it may be suspended to program on both main array and shadow row.

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.

Assumptions

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

2.6.5 ProgramVerify()

Description

This function checks if a programmed Flash range matches the corresponding source data buffer. In case of mismatch, the failed address, destination value and source value is saved and relevant error code is returned.

Prototype

```
UINT32 ProgramVerify (PSSD_CONFIG pSSDConfig,
   UINT32 dest,
   UINT32 size,
   UINT32 source,
   UINT32 *pFailAddress,
   UINT64 *pFailData,
   UINT64 *pFailSource,
   void (*CallBack)(void));
```

Arguments

Table 18. 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 double word boundary in main array or shadow row.
Size	Size, in byte, of the Flash region to verify.	If size = 0, C90FL_OK is returned. Its combination with *dest* should fall within either main array or shadow row.
Source	Verify source buffer address.	This address must reside on word boundary.
pFailAddress	Return first failing address in Flash.	Only valid when the function returns C90FL_ERROR_VERIFY.
pFailData	Returns first mismatch data in Flash.	Only valid when this function returns C90FL_ERROR_VERIFY.
pFailSource	Returns first mismatch data in buffer.	Only valid when this function returns C90FL_ERROR_VERIFY.
CallBack	Address of void call back function pointer.	Any addressable void function address. To disable it use NULL_CALLBACK macro.

Return values

Table 19. Return values for ProgramVerify()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_ALIGNMENT C90FL_ERROR_RANGE C90FL_ERROR_VERIFY

Troubleshooting

Table 20. Troubleshooting for ProgramVerify()

Returned error bits	Description	Solution
C90FL_ERROR_ALIGNMENT	This error indicates that *dest*/*size*/*source* isn't properly aligned	Check if *dest* and *size* are aligned on double word (64-bit) boundary. Check if *source* is aligned on word boundary
C90FL_ERROR_RANGE	The area specified by *dest* and *size* is out of the valid C90FL2 address range.	Check *dest* and *dest*+*size*, both should fall in the same C90FL2 address ranges, i.e. both in main array or both in shadow row
C90FL_ERROR_VERIFY	The content in C90FL2 and source data mismatch.	Check the correct source and destination addresses, erase the block and reprogram data into Flash.

Comments

The contents pointed by *pFailLoc*, *pFailData* and *pFailSource* are updated only when there is a mismatch between the source and destination regions.

Assumptions

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

2.6.6 CheckSum()

Description

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

Prototype

```
UINT32 CheckSum (PSSD_CONFIG pSSDConfig,
   UINT32 dest,
   UINT32 size,
   UINT32 *pSum,
   void (*CallBack)(void));
```

Arguments

Table 21. 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 double word boundary in either main array or shadow row.
Size	Size, in bytes, of the Flash region to check sum.	If size is 0 and the other parameters are all valid, C90FL_OK is returned. Its combination with *dest* should fall within either main array or shadow row.
pSum	Returns the sum value.	0x00000000 - 0xFFFFFFFF. Note that this value is only valid when the function returns C90FL_OK.
CallBack	Address of void call back function pointer.	Any addressable void function address. To disable it use NULL_CALLBACK macro.

Return values

Table 22. Return values for CheckSum()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_ALIGNMENT C90FL_ERROR_RANGE

Troubleshooting

Table 23. Troubleshooting for CheckSum()

Returned Error Bits	Description	Solution
C90FL_ERROR_ALIGNMENT	This error indicates that *dest*/*size* isn't properly aligned	Check if *dest* and *size* are aligned on double word (64-bit) boundary. Check if *source* is aligned on word boundary
C90FL_ERROR_RANGE	The area specified by *dest* and *size* is out of the valid C90FL2 address range.	Check *dest* and *dest+size*, both should fall in the same C90FL2 address ranges, i.e. both in main array or both in shadow row

Comments

None.

Assumptions

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

2.6.7 FlashSuspend()

Description

This function checks if there is any high voltage operation, erase or program, in progress on the C90FL2 module and if the operation can be suspended. This function suspends the ongoing operation if it can be suspended.

Prototype

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

Arguments

Table 24. 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 C90FL2 module after the function being called.	All return values are enumerated in *Table 27*.
suspendFlag	Return whether the suspended operation, if there is any, is suspended by this call.	TRUE: the operation is suspended by this call; FALSE: either no operation to be suspended or the operation is suspended not by this call.

Return values

Table 25. Return values for FlashSuspend()

Type	Description	Possible values
UINT32	Successful completion.	C90FL_OK

Troubleshooting

None.

Comments

After calling '*FlashSuspend()*', read is allowed on both main array space and shadow row without any Read-While-Write error. But data read from the blocks targeted for programming or erasing is indeterminate even if the operation is suspended.

This function should be used together with '*FlashResume()*'. The *suspendFlag* returned by '*FlashSuspend()*' determine whether '*FlashResume()*' needs to be called or not. If *suspendFlag* is TRUE, '*FlashResume()*' must be called symmetrically to resume the suspended operation.

Table 26 defines and describes various suspend states and associated suspend codes.

Table 26. SuspendState definitions

Argument	Code	Description	Valid operation after suspend
NO_OPERTION	0	There is no program/erase operation.	Erasing operation, programming operation and read are valid on both main array space and shadow row.
PGM_WRITE	1	There is a program sequence in interlock write stage.	Only read is valid on both main array space and shadow row.
ERS_WRITE	2	There is an erase sequence in interlock write stage.	Only read is valid on both main array space and shadow row.
ERS_SUS_PGM_WRITE	3	There is an erase-suspend program sequence in interlock write stage.	Only read is valid on both main array space and shadow row.
PGM_SUS	4	The program operation is in suspended state.	Only read is valid on both main array space and shadow row.
ERS_SUS	5	The erase operation on main array is in suspended state.	Programming operation is valid on both main arrayspace and shadow row. Read is valid on both main array space and shadow row.
SHADOW_ERS_SUS	6	The erase operation on shadow row is in suspended state.	Programming operation is valid on both main array space and shadow row. Read is valid on both main array space and shadow space.
ERS_SUS_PGM_SUS	7	The erase-suspended program operation is in suspended state.	Only read is valid on both main array space and shadow row.

Table 27 lists the Suspend Flag values returned against the Suspend State and the Flash block status.

Table 27. Suspending state and flag vs. C90FL2 status

suspendState	EHV	ERS	ESUS	PGM	PSUS	PEAS	suspendFlag
NO_OPERATION	X	0	X	0	X	X	FALSE
PGM_WRITE	0	0	X	1	0	X	FALSE
ERS_WRITE	0	1	0	0	X	X	FALSE
ESUS_PGM_WRITE	0	1	1	1	0	X	FALSE
PGM_SUS	1	0	X	1	0	X	TRUE
	X	0	X	1	1	X	FALSE
ERS_SUS	1	1	0	0	X	0	TRUE
	X	1	1	0	X	0	FALSE

Table 27. Suspending state and flag vs. C90FL2 status (continued)

suspendState	EHV	ERS	ESUS	PGM	PSUS	PEAS	suspendFlag
SHADOW_ERS_SUS	1	1	0	0	X	1	TRUE
	X	1	1	0	X	1	FALSE
ERS_SUS_PGM_SUS	1	1	1	1	0	X	TRUE
	X	1	1	1	1	X	FALSE

The values of EHV, ERS, ESUS, PGM, PSUS and PEAS represent the C90FL2 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 the '*FlashInit()*' API.

2.6.8 FlashResume()

Description

This function checks if there is any suspended erase or program operation on the C90FL2 module and resumes the suspended operation if there is any.

Prototype

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

Arguments

Table 28. 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 C90FL2 module after the function being called.	All return values are listed in *Table 30*.

Return values

Table 29. Return values for FlashResume()

Type	Description	Possible values
UINT32	Successful completion.	C90FL_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. It is better to call this function based on *suspendFlag* returned from '*FlashSuspend()*'.

Table 30 defines and describes various resume states and associated resume codes.

Assumptions

Table 30. resumeState definitions

Code Name	Value	Description
RES_NOTHING	0	No program/erase operation to be resumed
RES_PGM	1	A program operation is resumed
RES_ERS	2	A erase operation is resumed
RES_ERS_PGM	3	A suspended erase-suspended program operation is resumed

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

2.6.9 GetLock()

Description

This function checks the block locking status of Shadow/Low/Middle/High address spaces in the C90FL2 module.

Prototype

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

Arguments

Table 31. 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 and the block locking level, which determines the address space block locking register to be checked.	Refer to *Table 34* for valid values for this parameter.

Table 31. Arguments for GetLock() (continued)

Argument	Description	Range
blkLockEnabled	Indicate whether the address space block locking register is enabled for register writes	TRUE – The address space block locking register is enabled for register writes. FALSE – The address space block locking register is disabled for register writes.
blkLockState	Returns the blocks' locking status of indicated locking level in the given address space	Bit mapped value indicating the locking status of the specified locking level and address space. 1: The block is locked from program/erase. 0: The block is ready for program/erase

Return values

Table 32. Return values for GetLock()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_LOCK_INDICATOR

Troubleshooting

Table 33. Troubleshooting for GetLock()

Returned Error Bits	Possible causes	Solution
C90FL_ERROR_LOCK_INDICATOR	The input blkLockIndicator is invalid.	Set this argument to correct value listed in *Table 30*

Comments

Table 34 defines and describes various *blkLockIndicator* values.

Table 34. blkLockIndicator Definitions

Code name	Value	Description
LOCK_SHADOW_PRIMARY	0	Primary block lock protection of shadow address space
LOCK_SHADOW_SECONDARY	1	Secondary block lock protection of shadow address space
LOCK_LOW_PRIMARY	2	Primary block lock protection of low address space
LOCK_LOW_SECONDARY	3	Secondary block lock protection of low address space
LOCK_MID_PRIMARY	4	Primary block lock protection of mid address space
LOCK_MID_SECONDARY	5	Secondary block lock protection of mid address space
LOCK_HIGH	6	Block lock protection of high address space

For Shadow/Low/Mid address spaces, there are two block lock levels. The secondary level of block locking provides an alternative means to protect blocks from being modified. A logical "OR" of the corresponding bits in the primary and secondary lock registers for a block determines the final lock status for that block. For high address space there is only one block lock level.

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

The indicated address space determines the valid bits of *blkLockState*. The following diagrams show the block bitmap definitions of *blkLockState* for shadow/Low/Mid/High address spaces.

For low address space valid bits are from bit 0 to bit (*lowBlockNum – 1*). In which, *lowBlockNum* is the number of low blocks returned from FlashInit();

For middle address space valid bits are from bit 0 and bit (*midBlockNum – 1*). In which, *midBlockNum* is the number of middle blocks returned from FlashInit();

For high address space valid bits are from bit 0 to bit (*highBlockNum – 1*). In which, *highBlockNum* is the number of high blocks returned from FlashInit();

For shadow row valid bit is bit 0;

For example, below are bit allocations for blocks in Low/Mid/High Address Space of SPC56ELx:

blkLockState Bit Allocation for Shadow Address Space:

Table 35. blkLockState bit allocation for shadow address space

MSB **LSB**

bit 31	…	bit 1	bit 0
reserved	…	reserved	shadow row

Table 36. blkLockState bit allocation for low address space

MSB **LSB**

bit 31	…	bit 10	bit 9	bit 8	…	bit 1	bit 0
reserved	…	reserved	block 9	block 8	…	block 1	block 0

Table 37. blkLockState bit allocation for mid address space

MSB **LSB**

bit 31	…	bit 4	bit 3	bit 2	bit 1	bit 0
reserved	…	reserved	reserved	reserved	block 1	block 0

Table 38. blkLockState bit allocation for high address space

MSB **LSB**

bit 31	…	bit 6	bit 5	bit 4	…	bit 1	bit 0
reserved	…	reserved	block 5	block 4	…	block 1	block 0

Assumptions

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

2.6.10 SetLock()

Description

This function set the block lock state for Shadow/Low/Middle/High address space on the C90FL2 module to protect them from program/erase. The API provides password to enable block lock register writes when needed and write the block lock value to block lock register for the requested address space.

Prototype

```
UINT32 SetLock (PSSD_CONFIG pSSDConfig,
   UINT8 blkLockIndicator,
   UINT32 blkLockState,
   UINT32 password);
```

Arguments

Table 39. 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 34* 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 protection level and address space. 1: The block is locked from program/erase. 0: The block is ready for program/erase
password	A password is required to enable the block lock register for register write.	Correct passwords for block lock registers are 0xA1A1_1111 for Low/Mid Address Space Block Locking Register, 0xC3C3_3333 for Secondary Low/Mid Address Space Block Locking Register, and 0xB2B2_2222 for High Address Space Block Select Register.

Return values

Table 40. Return values for SetLock()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_LOCK_INDICATOR C90FL_ERROR_PASSWORD

Troubleshooting

Table 41. Troubleshooting for SetLock()

Returned error bits	Possible causes	Solution
C90FL_ERROR_LOCK_INDICATOR	The input blkLockIndicator is invalid.	Set this argument to correct value listed in *Table 34*.
C90FL_ERROR_PASSWORD	The given password cannot enable the block lock register for register writes.	Pass in a correct password.

Comments

The bit field allocation for *blkLockState* is same as that in ‘*GetLock()*’ function.

Assumptions

It assumes that the Flash block is initialized using the ‘*FlashInit()*’ API.

2.6.11 FlashDepletionRecover()

Description

This function recovers over-erased or depleted bits in Flash block. It is possible that a brownout during Flash erase operation leaves the bits in the Flash block(s) being erased at an over-erased or depleted state. Depending how depleted the bits are, the excessive column leakage caused by the bits might cause the following erase operation for brownout recovery to fail due to suppressed drain bias. For such case, this function is invoked to recover the depleted bits in those Flash block(s) so that they can be erased again for brownout recovery.

Prototype

```
UINT32 FlashDepletionRecover (PSSD_CONFIG pSSDConfig,
   BOOL shadowFlag,
   UINT32 lowEnabledBlocks,
   UINT32 midEnabledBlocks,
   UINT32 highEnabledBlocks,
   void (*CallBack)(void));
```

Arguments

Table 42. Arguments for FlashDepletionRecover()

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.
shadowFlag	Indicate either the main array or the shadow row to be recovered.	TRUE: the shadow row is recovered. The *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* are ignored; FALSE: The main array is recovered. Which blocks are erased in low, mid and high address spaces are specified by *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* respectively.
lowEnabledBlocks	To select the array blocks in low address space for recovering.	Bit-mapped value. Select the block in the low address space to be erased by setting 1 to the appropriate bit of *lowEnabledBlocks*. If there is not any block to be recovered in the low address space, *lowEnabledBlocks* must be set to 0.
midEnabledBlocks	To select the array blocks in mid address space for recovering.	Bit-mapped value. Select the block in the middle address space to be recovered by setting 1 to the appropriate bit of *midEnabledBlocks*. If there is not any block to be recovered in the middle address space, *midEnabledBlocks* must be set to 0.
highEnabledBlocks	To select the array blocks in high address space for recovering.	Bit-mapped value. Select the block in the high address space to be recovered by setting 1 to the appropriate bit of *highEnabledBlocks*. If there is not any block to be recovered in the high address space, *highEnabledBlocks* must be set to 0.
CallBack	Address of void call back function pointer.	Any addressable void function address. To disable it, use NULL_CALLBACK macro.

Return values

Table 43. Return values for FlashDepletionRecover ()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_BUSY C90FL_ERROR_EGOOD

Troubleshooting

Table 44. Troubleshooting for FlashDepletionRecover ()

Error codes	Possible causes	Solution

C90FL_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 (FLASH_MCR-ERS is high);
		program in progress (FLASH_MCR-PGM is high);

C90FL_ERROR_ EGOOD	Depletion recovery failed.	Check if the C90FL2 is available and high voltage is applied to C90FL2. Then try to do the recovery operation again.

Comments

Refer to this section of FlashErase() function.

Assumptions

It assumes that the Flash block is initialized using the *'FlashInit()'* API. User provides the correct ssdConfig parameters to *FlashDepletionRecover()* as returned by *FlashInit()*.

2.7 User test mode functions

2.7.1 FlashArrayIntegrityCheck()

Description

This function checks the array integrity of the Flash. 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.

Prototype

```
UINT32 FlashArrayIntegrityCheck (PSSD_CONFIG pSSDConfig,
   UINT32 lowEnabledBlocks,
   UINT32 midEnabledBlocks,
   UINT32 highEnabledBlocks,
   BOOL addrSeq,
   MISR misrValue,
   void (*CallBack)(void));
```

Arguments

Table 45. 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 erasing.	Bit-mapped value. Select the block in the low address space whose array integrity is to be evaluated by setting 1 to the appropriate bit of *lowEnabledBlocks*. If there is not any block to be evaluated in the low address space, *lowEnabledBlocks* must be set to 0.
midEnabledBlocks	To select the array blocks in mid address space for erasing.	Bit-mapped value. Select the block in the middle address space whose array integrity is to be evaluated by setting 1 to the appropriate bit of *midEnabledBlocks*. If there is not any block to be evaluated in the middle address space, *midEnabledBlocks* must be set to 0.
highEnabledBlocks	To select the array blocks in high address space for erasing.	Bit-mapped value. Select the block in the high address space whose array integrity is to be evaluated by setting 1 to the appropriate bit of *highEnabledBlocks*. If there is not any block to be evaluated in the high address space, *highEnabledBlocks* must be set to 0.
addrSeq	To determine the address sequence to be used during array integrity checks.	The default sequence (*addrSeq* = 0 (C90FL_ABF_SEQ)) is meant to replicate sequences normal “user” code follows, and thoroughly check the read propagation paths. This sequence is proprietary. The alternative sequence (*addrSeq* = 1 (C90FL_LOG_SEQ)) 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.
misrValue	A structure variable containing the MISR values calculated by the user using the offline MISR generation tool.	The individual MISR words can range from 0x00000000 - 0xFFFFFFFF
CallBack	Address of void call back function pointer.	Any addressable void function address. To disable it use NULL_CALLBACK macro.

Return values

Table 46. Return values for FlashArrayIntegrityCheck()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_AIC_MISMATCH C90FL_ERROR_AIC_NO_BLOCK

Troubleshooting

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

Table 47. Troubleshooting for FlashArrayIntegrityCheck()

Returned error bits	Possible causes	Solution
C90FL_ERROR_AIC_MISMATCH	The MISR value calculated by the user is incorrect.	Re-calculate the MISR values using the correct Data and addrSeq.
	The MISR calculated by the Hardware is incorrect.	Hardware Error.
C90FL_ERROR_AIC_NO_BLOCK	None of the Blocks are enabled for Array Integrity Check	Enable any of the blocks using variables *lowEnabledBlocks, midEnabledBlocks* and *highEnabledBlock.*

Comments

The inputs *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* are bit-mapped arguments that are used to select the blocks to be evaluated in the Low/Mid/High 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* or *highEnabledBlocks*.

The bit allocations for blocks in one address space are: bit 0 is assigned to block 0, bit 1 to block 1, etc. The following diagrams show the formats of *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* for the C90FL2 module.

For low address space valid bits are from bit 0 to bit (*lowBlockNum – 1*). In which, *lowBlockNum* is the number of low blocks returned from FlashInit();

For middle address space valid bits are from bit 0 and bit (*midBlockNum – 1*). In which, *midBlockNum* is the number of middle blocks returned from FlashInit();

For high address space valid bits are from bit 0 to bit (*highBlockNum – 1*). In which, *highBlockNum* is the number of high blocks returned from FlashInit();

For example, below are bit allocations for blocks in Low/Mid/High Address Space of SPC56ELx:

Bit Allocation for Blocks in Low Address Space:

Table 48. Bit allocation for blocks in low address space

MSB **LSB**

bit 31	…	bit 10	bit 9	bit 8	…	bit 1	bit 0
reserved	…	reserved	block 9	block 8	…	block 1	block 0

Table 49. Bit allocation for blocks in middle address space

MSB LSB

bit 31	…	bit 4	bit 3	bit 2	bit 1	bit 0
reserved	…	reserved	reserved	reserved	block 1	block 0

Table 50. Bit allocation for blocks in high address space

MSB LSB

bit 31	…	bit 6	bit 5	bit 4	…	bit 1	bit 0
reserved	…	reserved	block 5	block 4	…	Block 1	Block 0

If no blocks are enabled the C90FL_ERROR_AIC_NO_BLOCK error code is returned.

Depending on the address sequence specified the MISR values are calculated for the enabled blocks using the corresponding sequence. If the MISR values calculated by the hardware is not the same as the values passed to this API by the user then the API returns the error code C90FL_ERROR_AIC_MISMATCH.

Assumptions

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

2.7.2 FlashECCLogicCheck()

Description

This function checks the ECC logic of the Flash. The API simulates a single or double bit fault depending on the user input. If the simulated ECC error is not detected, then the error code C90FL_ERROR_ECC_LOGIC is returned.

Prototype

```
UINT32 FlashECCCLogicCheck (PSSD_CONFIG pSSDConfig,
   UINT64 dataVal,
   UINT64 errBits,
   UINT32 eccValue);
```

Arguments

Table 51. Arguments for FlashECCLogicCheck()

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.
dataValue	The 64 bits of data for which the ECC is calculated. The bits of *dataValue* are flipped to generate single or double bit faults.	Any 64-bit value.

Table 51. Arguments for FlashECCLogicCheck() (continued)

Argument	Description	Range
errBits	Is a 64-bit mask of the bits at which the user intends to inject error.	Any 64-bit value, except zero.
eccValue	It’s a 32 bit value which has to be passed by user. This is calculated by using an offline ECC Calculator.	This is a corresponding ECC value for the data value passed by the user. Note: Same data words should be used in offline ECC calculator and Flash ECC logic check API.

Return values

Table 52. Return values for FlashECCLogicCheck()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_ECC_LOGIC

Troubleshooting

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

Table 53. Troubleshooting for FlashECCLogicCheck()

Returned error bits	Possible causes	Solution
C90FL_ERROR_ECC_LOGIC	The ECC value calculated by the user is incorrect.	Re-calculate the ECC values using the correct Data.
	Hardware is failure	Hardware Error.

Comments

Depending on the *errBits* value, a single or double bit faults are simulated. When a Flash read is done, if the simulated error has not occurred, then the API returns the error code C90FL_ERROR_ECC_LOGIC.

Assumptions

It assumes that the Flash block is initialized using the ‘*FlashInit()*’ API.

2.7.3 FactoryMarginReadCheck()

Description

This function checks the Factory Margin reads of the Flash. 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 is compared to the values passed by the user, if they are not the same, then an error code is returned.

Prototype

```
UINT32 FactoryMarginReadCheck (PSSD_CONFIG pSSDConfig,
   UINT32 lowEnabledBlocks,
   UINT32 midEnabledBlocks,
   UINT32 highEnabledBlocks,
   BOOL marginLevel,
   MISR misrValue,
   void (*CallBack)(void));
```

Arguments

Table 54. Arguments for FactoryMarginReadCheck()

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 erasing.	Bit-mapped value. Select the block in the low address space whose array integrity is to be evaluated by setting 1 to the appropriate bit of *lowEnabledBlocks*. If there is not any block to be evaluated in the low address space, *lowEnabledBlocks* must be set to 0.
midEnabledBlocks	To select the array blocks in mid address space for erasing.	Bit-mapped value. Select the block in the middle address space whose array integrity is to be evaluated by setting 1 to the appropriate bit of *midEnabledBlocks*. If there is not any block to be evaluated in the middle address space, *midEnabledBlocks* must be set to 0.
highEnabledBlocks	To select the array blocks in high address space for erasing.	Bit-mapped value. Select the block in the high address space whose array integrity is to be evaluated by setting 1 to the appropriate bit of *highEnabledBlocks*. If there is not any block to be evaluated in the high address space, *highEnabledBlocks* must be set to 0.
marginLevel	To determine the margin level to be used during factory margin read checks.	Selects the margin level that is being checked. Margin can be checked to an erased level (marginLevel=1 (C90FL_FMR_ERASED_LEVEL)) or to a programmed level (marginLevel = 0 (C90FL_FMR_PROGRAMMED_LEVEL)).
misrValue	A structure variable containing the MISR values calculated by the user using the offline MISR generation tool.	The individual MISR words can range from 0x00000000 - 0xFFFFFFFF
CallBack	Address of void call back function pointer.	Any addressable void function address. To disable it use NULL_CALLBACK macro.

Return values

Table 55. Return values for FactoryMarginReadCheck()

Type	Description	Possible values
UINT32	Successful completion or error value.	C90FL_OK C90FL_ERROR_FMR_MISMATCH C90FL_ERROR_FMR_NO_BLOCK

Troubleshooting

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

Table 56. Troubleshooting for FactoryMarginReadCheck()

Returned error bits	Possible causes	Solution
C90FL_ERROR_FMR_MISMATCH	The MISR value calculated by the user is incorrect.	Re-calculate the MISR values using the correct Data and address.
	The MISR calculated by the Hardware is incorrect.	Hardware Error.
C90FL_ERROR_FMR_NO_BLOCK	None of the Blocks are enabled for Factory Margin Read Check	Enable any of the blocks using variables *lowEnabledBlocks, midEnabledBlocks* and *highEnabledBlock.*

Comments

The inputs *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* are bit-mapped arguments that are used to select the blocks to be evaluated in the Low/Mid/High 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* or *highEnabledBlocks*.

The bit allocations for blocks in one address space are: bit 0 is assigned to block 0, bit 1 to block 1, etc. The following diagrams show the formats of *lowEnabledBlocks*, *midEnabledBlocks* and *highEnabledBlocks* for the C90FL2 module.

For low address space valid bits are from bit 0 to bit (*lowBlockNum – 1*). In which, *lowBlockNum* is the number of low blocks returned from FlashInit();

For middle address space valid bits are from bit 0 and bit (*midBlockNum – 1*). In which, *midBlockNum* is the number of middle blocks returned from FlashInit();

For high address space valid bits are from bit 0 to bit (*highBlockNum – 1*). In which, *highBlockNum* is the number of high blocks returned from FlashInit();

For example, below are bit allocations for blocks in Low/Mid/High Address Space of SPC56ELx:

Bit Allocation for Blocks in Low Address Space:

Table 57. Bit allocation for blocks in low address space

MSB **LSB**

bit 31	...	bit 10	bit 9	bit 8	...	bit 1	bit 0
reserved	...	reserved	block 9	block 8	...	block 1	block 0

Table 58. Bit allocation for blocks in middle address space

MSB **LSB**

bit 31	...	bit 4	bit 3	bit 2	bit 1	bit 0
reserved	...	reserved	reserved	reserved	block 1	block 0

Table 59. Bit allocation for blocks in high address space

MSB **LSB**

bit 31	...	bit 6	bit 5	bit 4	...	bit 1	bit 0
reserved	...	reserved	block 5	block 4	...	Block 1	Block 0

If no blocks are enabled the C90FL_ERROR_FMR_NO_BLOCK error code is returned.

The MISR values are calculated for the enabled blocks using the logical sequence. If the MISR values calculated by the hardware is not the same as the values passed to this API by the user then the API returns the error code C90FL_ERROR_FMR_MISMATCH.

Assumptions

It assumes that the Flash block is initialized using the *‘FlashInit()’* API.

Appendix A CallBack timings

Table 60. CallBack timings period for SPC56ELx

API name	Time (µs) System clock = 40MHz
FlashProgram() (size = 0x1000)	2.250
ProgramVerify()(size = 0x1000, CALLBACK_PV = 70)	96.825
FlashErase() (low block 0)	2.350
BlankCheck() (size = LOW_BLOCK0_SIZE, CALLBACK_BC = 75)	99.250
CheckSum (size = 0x1000, CALLBACK_CS = 105)	107.425
FlashArrayIntegrityCheck (low block 0)	2.25
FactoryMarginReadCheck (low block 0)	2.35

Note: *Callback time period for ‘CheckSum()’ is measured with CALLBACK_CS (CallBack function period for checksum)*

Callback time period for ‘ProgramVerify()’ is measured with CALLBACK_PV (CallBack function period for program verify)

Callback time period for ‘BlankCheck()’ is measured with CALLBACK_BC (CallBack function period for program verify)

Appendix B System requirements

The C90FL2 SSD is designed to support a single C90FL2 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 structure.

Table 61. System requirements

Tool name	Description	Version number
CodeWarrior IDE	Development tool	2.6
Diab PowerPC compiler	Compiler	5.5.1
GAP tool	Debugger	1.5
GreenHills	Development tool	5.1.6C
LauterBach Trace32	Debugger	

Appendix C Acronyms

Table 62. Acronyms

Abbreviation	Complete name
API	Application Programming Interface
BIU	Bus Interface Unit
ECC	Error Correction Code
SSD	Standard Software Driver

Appendix D Documentation References

1. SPC56EL60 32-bit MCU family built on the embedded Power Architecture®
(RM0032, Doc ID 15265)

Revision history

Table 63. Document revision history

Date	Revision	Changes
17-May-2013	1	Initial release.
17-Sep-2013	2	Updated Disclaimer.
13-Jul-2020	3	Updated title.