Freescale Semiconductor
Application Note

Document Number: AN4860
Rev. 1, 11/2014

Flash Driver Library for MC56F847xx and MC56F827xx DSC Family

by Zbynek Mynar

Contents

1 Introduction

Applications that require stored data to be available even after restart or power loss must utilize non-volatile memory. Using an extra EEPROM part is not cost-effective. The internal flash memory of a Digital Signal Controller (DSC) is a better choice. The disadvantage of this approach is that unlike a read operation, the write and erase operations of the Flash memory are more complex and cannot be done using simple memory bus access. The Flash Driver Library provides a simple way of performing all basic Flash memory operations and introduces new incremental writing and flash over FreeMASTER features.

This application note addresses:

- Flash memory and its interface on the MC56F847xx and MC56F827xx DSC families
- Flash Driver Library functions and features
- Installation of the Flash Driver Library

2 Supported digital signal controllers

The key features of MC56F847xx DSC family are:

- Up to 100 MHz program execution from flash memory and RAM
- Up to 256 KB (128 KW) program flash memory with sector size of 2 KB (1 KW)
- Up to 32 KB (16 KW) data flash memory with sector size of 1 KB (512 W)
- Flash Memory Module (FTFL) periphery that provides:
  - Automated, built-in, program and erase algorithms with verification
  - Read access to program flash memory possible while programming or erasing data in the data flash memory

The MC56F847xx devices also feature a FlexMemory module that provides enhanced EEPROM (EEE) functionality. It is not used in the current version of the Flash Driver Library. For more information see the *MC56F847xx Reference Manual* or AN4689 on freescale.com.

The MC56F827xx DSC family features:

- Up to 50 MHz program execution from flash memory and 100 MHz from RAM
- Access of data flash (using data address) is 25 MHz
- Up to 64 KB (32 KW) program flash memory with sector size of 1 KB (512 W)
- Flash Memory Module (FTFA) periphery that enables automated, built-in, program and erase algorithms with verification

2.1 Memory maps

For the informed use of the following sections, it is necessary to be familiar with the memory maps of both DSC families.

All supported DSCs are based on modified Harvard architecture, therefore there are two memory address spaces: one for program and one for data. However, all physical memories are mapped into both address spaces. When using assembler instruction, the program and data memory address spaces are differentiated by X: and P: prefixes, where X: is for data and P: is for program address space. Figure 1 shows the memory map of MC56F84789 DSC and Figure 2 shows the memory map of MC56F82748 DSC. Both DSCs represent the largest models (in terms of memory) of their families. For an explanation of Large Data Model (LDM) and Small Data Model (SDM) see Section 3, “DSC Flash memory description.”

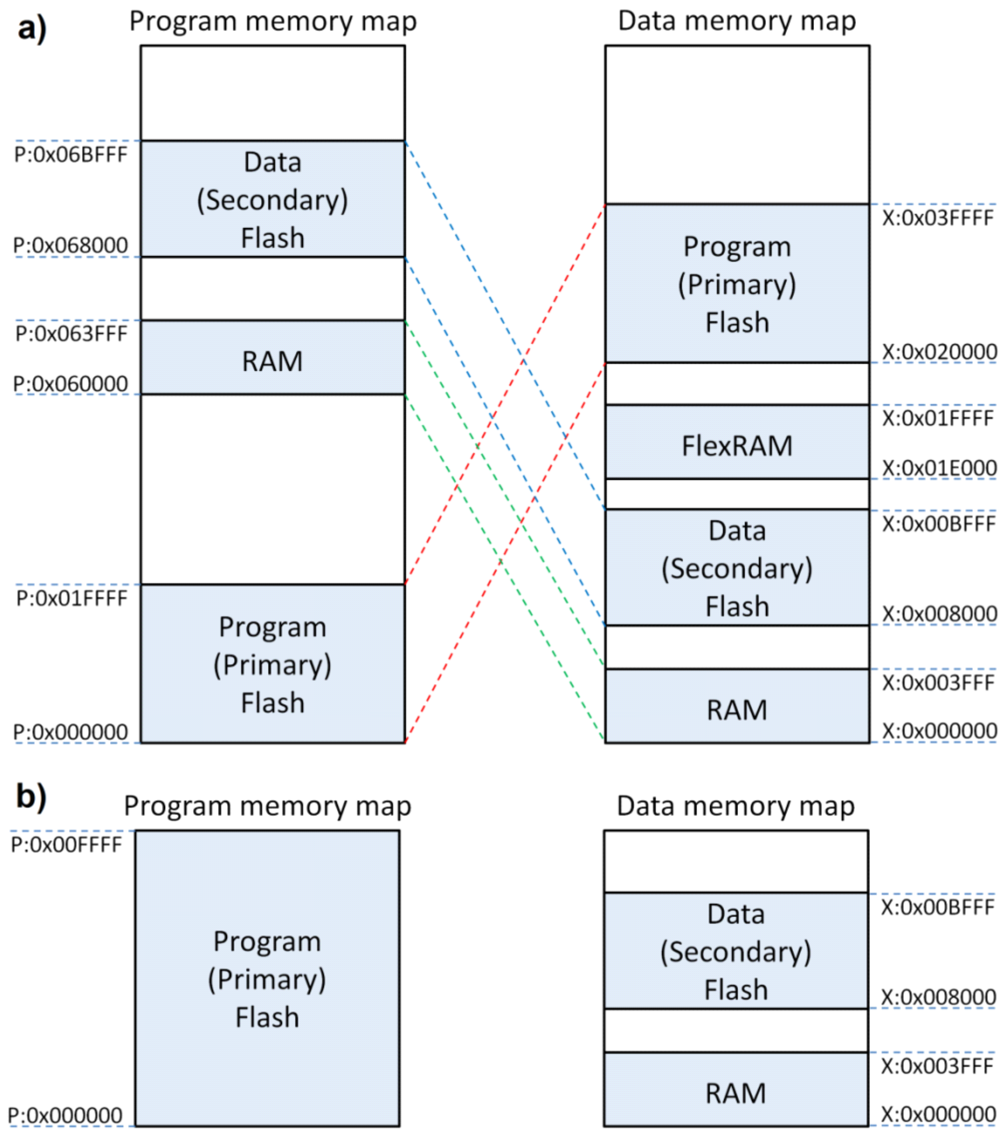

Figure 1. Memory map of MC56F84789 for a) large data model and b) small data model

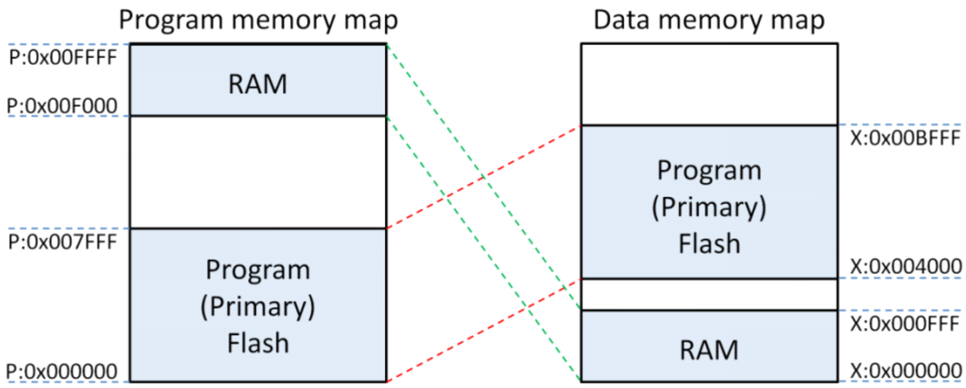

Figure 2. Memory map of MC56F82748 for small and large data model

3 DSC Flash memory description

Flash memory generally allows fast data access, while writing and erasing is significantly slower. It is not possible to write or erase a single byte. These operations can only be conducted on larger memory areas. The smallest erasable memory block is called sector. An erased bit reads '1' and a programmed bit reads '0'. The programming operation is unidirectional; this means that bits can be only toggled from the '1' state (erased) to the '0' state (programmed). Only the erase operation restores bits from '0' to '1'. Cumulative programming of bits (reprogramming without an erase) is not allowed as this overstresses the device.

To modify flash memory content, both supported DSC families contain a Flash Memory Module periphery (FTFL at MC56F847xx devices and FTFA at MC56F827xx devices). In terms of basic flash operations provided by Flash Driver Library, both FTFL and FTFA are fully compatible (exceptions will be mentioned). This application note describes only the FTFL.

To read data stored on flash memory, a simple memory bus access is required (the same way data in RAM are accessed). However, it should be noted that in the case of using C programming language on the MC56F847xx DSC family, the flash memory address might reach beyond 64 KB (the address limit when using Small Data Model). This can be solved by using Large Data Model (LDM), which uses 24-bit (instead of 16-bit) pointers and consequently more memory and processor time. To eliminate the need for LDM, the Flash Driver Library provides a special read function.

3.1 Flash memory module

All flash functions provided by FTFL require the user to set up and launch a flash command through a series of peripheral bus writes. The register serving this purpose is the FTFL_FSTAT byte register that provides control and status information and eight FTFL_FCCOB byte registers which serve as a flash command buffer.

To launch a general flash command, the following procedure must be performed:

1. First, the ACCERR and FPVIOL bits in the FSTAT register must be zero and the CCIF flag must read one to verify that any previous commands have finished. If CCIF is zero, a previous command is still being executed, a new command write sequence cannot be started, and all writes to the FCCOB registers are ignored.
2. The user must load the FCCOB registers with all parameters required by the desired flash command. The individual registers that make up the FCCOB data set can be written in any order.
3. Once all relevant command parameters have been loaded, the user launches the command by clearing the FSTAT[CCIF] bit by writing a '1' to it. The CCIF flag remains zero until the flash command completes.
4. The FTFL prevents a new command from launching (can't clear CCIF) if the previous command resulted in an access error (FSTAT[ACCERR]=1) or a protection violation (FSTAT[FPVIOL]=1). In error scenarios, two writes to FSTAT are required to initiate the next command: the first write clears the error flags, the second write clears CCIF.

Generic command write sequence is illustrated in Figure 3.

Once the flash command is written, the FTFL begins with processing:

1. The flash memory module reads the command code and performs a series of parameter and protection checks. If the parameters check fails, the FSTAT[ACCERR] (access error) flag is set. Usually, access errors suggest that the command was not set up with valid parameters in the FCCOB register group. Program and erase commands also check the address to determine, whether the operation is requested to execute on protected areas. If the protection check fails, the FSTAT[FPVIOL] (protection error) flag is set. If the parameter or protection checks fail, a command processing is terminated and the FSTAT[CCIF] bit is set.
2. The command proceeds to execution. Run-time errors, such as failure to erase verify, may occur. These errors are reported in the FSTAT[MGSTAT0] bit.
3. Command execution results, if applicable, are reported back to the user via the FCCOB and FSTAT register.
4. The FSTAT[CCIF] bit is set, after the FTFL completes flash command.

For more details, see the MC56F847xx or MC56F827xx reference manuals at freescale.com.

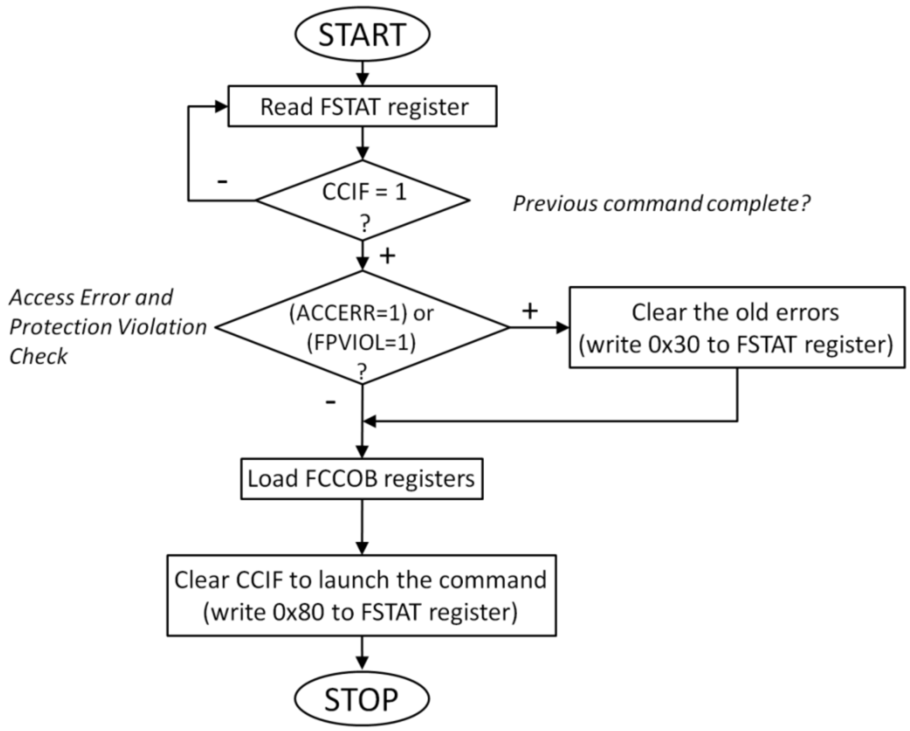

Figure 3. Generic Flash Command Launch Sequence Flowchart

3.2 FTFL commands

All flash commands used by the Flash Driver Library are listed in Table 1.

Table 1. List of flash commands used by Flash Driver Library

Command Name	Code	Function
Read 1s Section	0x01	Verify that a given number of flash locations (longwords or phrases) from a starting address are erased (Blank Check)
Program Longword	0x06	Program four bytes into flash
Erase Flash Sector	0x09	Erase all bytes in a flash sector

It is not possible to run multiple flash memory operations on a single flash memory module simultaneously, for example trying to read from primary flash while writing to it. Therefore, the code should be executed from RAM. This is discussed further in Section 5, “Installation guide.” However, because MC56F847xx DSC family has two flash memory modules (primary and secondary), a few cases of simultaneous operations are allowed (see Table 2).

Table 2. Allowed simultaneous memory operations on MC56F847xx DSC family

		Primary Flash			Secondary Flash		
		Read	Write	Erase	Read	Write	Erase
Primary Flash	Read	x				OK	OK
	Write		x		OK		
	Erase			x	OK		
Secondary Flash	Read		OK	OK	x		
	Write	OK				x	
	Erase	OK					x

The addresses used by the FTFL differ from those specified in Section 2.1, “Memory maps.” The FTFL flash addresses are byte-oriented and always start from zero. In the MC56F847xx DSC family, the secondary and primary flash addresses are distinguished by bit 23 of the address, where '0' is for primary flash and '1' is for secondary flash. An example of flash address generation is shown in Figure 4.

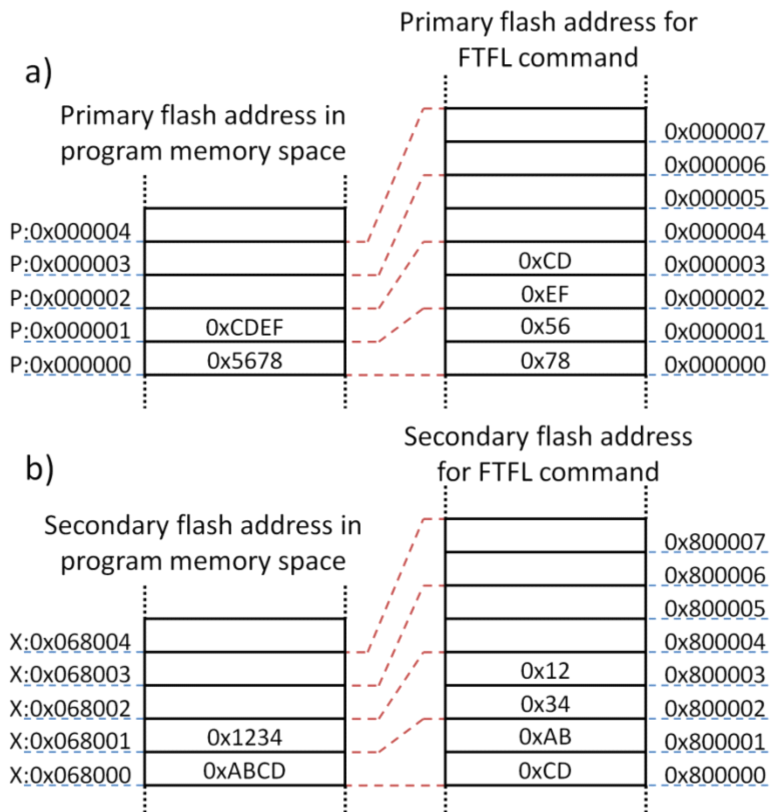

Figure 4. FTFL address generation example for a) primary flash; b) secondary flash

3.2.1 Blank check command

The “Read 1s Section” command checks to see, whether a section of a program or data flash memory is erased. This command requires the starting address and the number of phrases or longwords to be verified. The margin choice parameter allows a user to apply a non-standard read reference level. This could be used for special diagnostic routines. For standard operations it should be set to 0x00 (normal margin level). Flash command parameters differ slightly, depending on which flash memory is accessed (see Table 3 and Table 4).

Table 3. Read 1s section command FCCOB requirements (primary flash of MC56F847xx family)

FCCOB Number	FCCOB Contents [7:0]
0	0x01 (Read 1s Section Command Code)
1	Flash address [23:16] of the first phrase to be verified
2	Flash address [15:8] of the first phrase to be verified
3	Flash address [7:0]* of the first phrase to be verified
4	Number of phrases to be verified [15:8]
5	Number of phrases to be verified [7:0]
6	Read-1 Margin Choice

Note: * Flash address must be phrase aligned (Flash address [2:0] = 000)

Table 4. Read 1s section command FCCOB requirements (secondary flash of MC56F847xx and flash of MC56F827xx DSC family)

FCCOB Number	FCCOB Contents [7:0]
0	0x01 (Read 1s Section Command Code)
1	Flash address [23:16] of the first longword to be verified
2	Flash address [15:8] of the first longword to be verified
3	Flash address [7:0]* of the first longword to be verified
4	Number of longwords to be verified [15:8]
5	Number of longwords to be verified [7:0]
6	Read-1 Margin Choice

Note: * Flash address must be longword aligned (Flash address [1:0] = 00)

If the flash memory area specified by parameters is not erased, the FSTAT[MGSTAT0] bit is set once the command finishes. Possible flash command errors are listed in Table 5.

Table 5. Read 1s section command error handling

Error Condition	Error Bit
Command not available in current mode/security	FSTAT[ACCERR]
An invalid margin code is supplied	FSTAT[ACCERR]
An invalid flash address is supplied	FSTAT[ACCERR]
Flash address is not longword aligned	FSTAT[ACCERR]
The requested section crosses a Flash block boundary	FSTAT[ACCERR]
The requested number of phrases/longwords is zero	FSTAT[ACCERR]
Read-1s fails (memory area is not erased)	FSTAT[MGSTAT0]

3.2.2 Program longword command

The "Program Longword" command programs four previously erased bytes in the flash memory using an embedded algorithm. The command parameters are listed in Table 6.

Table 6. Program longword command FCCOB requirements

FCCOB Number	FCCOB Contents [7:0]
0	0x06 (Program Longword Command Code)
1	Flash address [23:16]
2	Flash address [15:8]
3	Flash address [7:0]*
4	Byte 0 program value (written to the supplied address + 0b11)
5	Byte 1 program value (written to the supplied address + 0b10)
6	Byte 2 program value (written to the supplied address + 0b01)
7	Byte 3 program value (written to the supplied address)

Note: * Flash address must be longword aligned (Flash address [1:0] = 00)

The command has built-in automated verification of written data. Possible flash command errors are listed in Table 7.

Table 7. Program longword command error handling

Error Condition	Error Bit
Command not available in current mode/security	FSTAT[ACCERR]
An invalid flash address is supplied	FSTAT[ACCERR]
Flash address is not longword aligned	FSTAT[ACCERR]
Flash address points to a protected area	FSTAT[FPVIOL]
Any errors have been encountered during the verify operation	FSTAT[MGSTAT0]

3.2.3 Erase flash sector command

The “Erase Flash Sector” command erases all addresses in a flash sector. The command parameters are listed in Table 8.

Table 8. Erase flash sector command FCCOB requirements

FCCOB Number	FCCOB Contents [7:0]
0	0x09 (Erase Flash Sector Command Code)
1	Flash address [23:16]
2	Flash address [15:8]
3	Flash address [7:0]

The command has built-in automated erase-verification. Possible flash command errors are listed in Table 9.

Table 9. Erase flash sector command error handling

Error Condition	Error Bit
Command not available in current mode/security	FSTAT[ACCERR]
An invalid flash address is supplied	FSTAT[ACCERR]
Flash address is not longword aligned	FSTAT[ACCERR]
Flash address points to a protected area	FSTAT[FPVIOL]
Any errors have been encountered during the verify operation	FSTAT[MGSTAT0]

4 Flash Driver Library

The Flash Driver Library was developed in order to provide a driver with a simple API that would perform all basic flash memory operations as described in Section 3, “DSC Flash memory description.”
New features of incremental writing and flash over FreeMASTER were also introduced (see the following sections).

4.1 Incremental writing feature

This feature allows frequent writing of constant-sized entries into a designated flash memory area. The new entries are added behind previous ones, rather than erasing memory every time. This improves flash memory endurance and allows access to older entries. Once the designated memory is full, older entries are erased. Size of erased flash memory area depends on the selected mode:

“Erase All” mode – entire designated flash memory is erased and no previous entries can be accessed. This mode is simpler and therefore incremental writing functions are slightly faster.

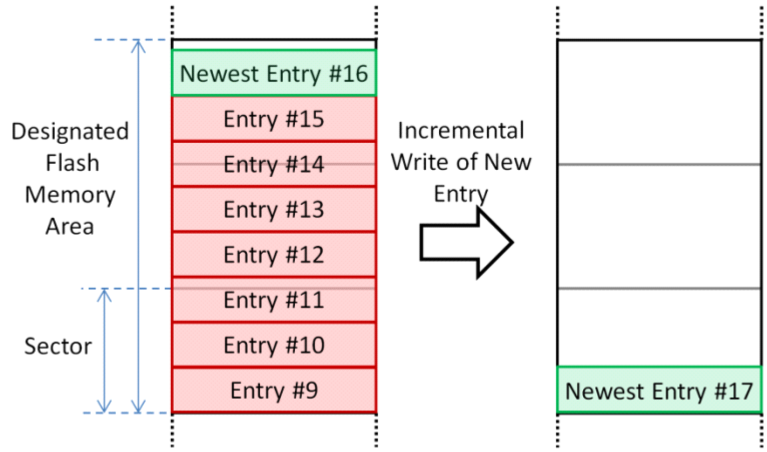

Figure 5. Erasing of previous entries with "Erase All" mode

“Erase Sector” mode – only a single sector is erased. This allows access to older entries all the time (if more than one sector is dedicated for incremental writing). A new entry that would cross the border of two sectors is aligned to the beginning of the next sector and empty space is left at the end of previous sector.

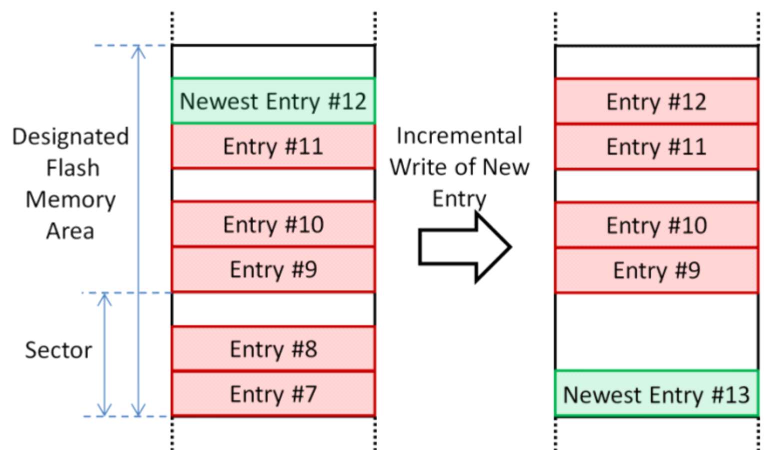

Figure 6. Erasing of previous entries with "Erase Sector" mode

Prior to compilation, the user can modify the structure of an entry freely. The only restriction is that the entry cannot have a size bigger than one sector. One entry structure variable is kept in RAM at all times. At the end of the initialization of the Flash Driver Library (see Section 4.3.5, “FlashDrv_Init()”), this variable is loaded with data from the last entry. The user can make changes in this structure variable, including loading it with data from an older entry, using one of FDL functions. To create a new entry (store the structure variable into flash), the user simply calls the appropriate FDL function.

The designated flash memory area must not be used for any other purpose, as writing to this memory area may lead to unexpected results.

All Flash Driver Library settings, including type definitions, are done in the `FlashDrv_cfg.h` file. See Section 4.4, "Flash Driver Library configuration" for more details.

4.2 Flash over FreeMASTER feature

FreeMASTER is a real-time debug monitor and data visualization tool. It supports non-intrusive monitoring as well as modification of target variables in real-time. FreeMASTER is composed of PC application and target side software (see freescale.com for more information).

Among other offerings, FreeMASTER provides a Flash Programming feature. This GUI communicates with a target-side FDL function using simple memory writes and read accesses. This function then calls on other FDL functions to perform desired flash memory operation. Two global variables are allocated in embedded application. One is command status word used to signal data-ready and operation-completed status. The other variable is data buffer of user-defined size for data and parameters exchange.

4.2.1 Flash programming interface description

FreeMASTER's Flash Programming GUI is shown in Figure 7. To open it go Tools > Flash Programming. It is available only after a successful connection to the target.

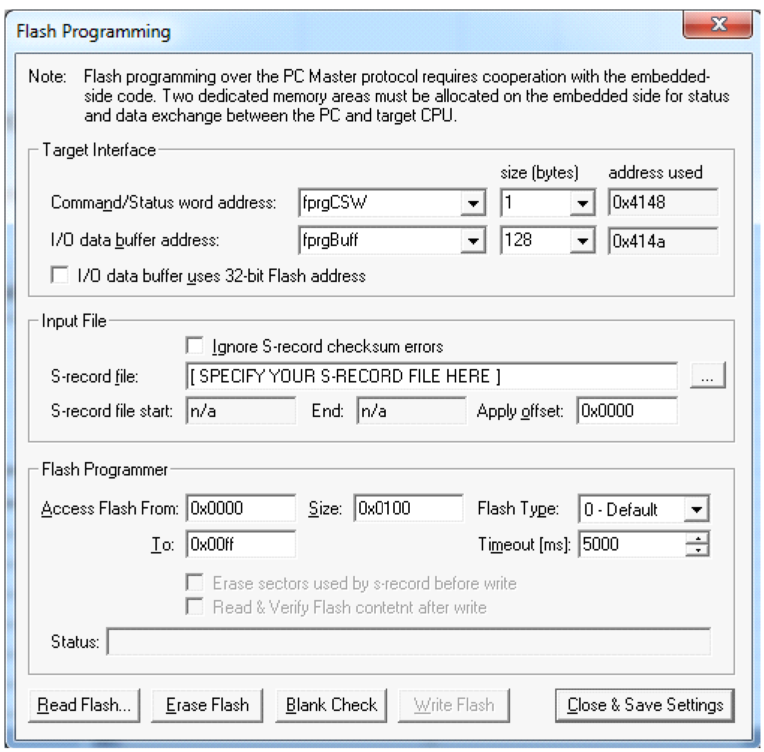

Figure 7. Flash programming GUI window

The "Target Interface" box specifies communication parameters. The command status word and data buffer names must be kept as default (`fprgCSW` and `fprgBuff`). The user can specify the size of data buffer.

However, the same or bigger size must be set in embedded application (see `FLASHDRV_FMSTR_BUFF_SIZE` define in Section 4.4, “Flash Driver Library configuration”). The checkbox determines whether a 32-bit or 16-bit address is used.

In the “Input File” box, the source S-record file for write operation can be selected. The S-record is a well-known file format that stores binary information in the ASCII hex text form.

The “Flash Programmer” box serves to set the address range, maximal communication timeout, and whether to use erase before write and read after write features. Flash Type selection is not supported in the current version of FDL and has no effect.

The Read Flash and Blank Check features use addresses specified in the “Flash Programmer” box. The Erase Flash feature uses this addresses as well. However, all sectors covered by selected flash memory area are erased (see Section 4.3.2, “FlashDrv_EraseSectors()”). The Read Flash feature saves data from flash memory area to S-record file. The Write Flash button programs all data from S-record file selected in “Input File” box into flash memory.

All addresses and data sizes specified in Flash Programming GUI are byte-oriented. However, all FDL functions are word-oriented. Therefore, passing any word misaligned address or data size will result in failure.

4.3 User-available Flash Driver Library functions

A complete list of FDL functions can be found in Table 10. All functions except `FlashDrv_Read()` were written in C language.

Table 10. List of Flash Driver Library User-available Functions

Function Name	Short Description
FlashDrv_BlankCheck()	Flash memory area blank check
FlashDrv_EraseSectors()	Erase sectors function
FlashDrv_Write()	Programming flash memory area function
FlashDrv_Read()	Copy data from flash memory
FlashDrv_Init()	Flash Driver Library initialization function
FlashDrv_IncWrite()	Incremental write of new entry
FlashDrv_GetEntry()	Get data from older entry
FlashDrv_FMSTRpoll()	Flash over FreeMASTER poll function

All functions use the following basic data types:

- UWord8 – unsigned byte. Range: <0, 255>
- UWord16 – unsigned word (two bytes). Range: <0, 2^{16})
- UWord32 – unsigned longword (four bytes). Range: <0, 2^{32})

The return codes are defined in `FlashDrv.h` file. See the following table for their list.

Table 11. FDL functions return codes

Return Code	Defined Value
FLASHDRV_SUCCESS	0
FLASHDRV_FAIL	1
FLASHDRV_ACCESS_ERROR	2
FLASHDRV_PROT_VIOLATION	3

The Flash Driver Library API is word-oriented, so it corresponds with MPU natural addressing. All functions use program address space addresses (see Section 3.1, “Flash memory module”).

The execution time of FDL functions was measured for optimization level four in CodeWarrior 10.4.

4.3.1 FlashDrv_BlankCheck()

This function encapsulates the Read 1s FTFL flash command (see section Section 3.2.1, “Blank check command”) Although this function accepts word-oriented parameters, the memory area checked is always longword-aligned. The reason for this is that it is only possible to program whole longword. Therefore, successful blank check of word-aligned flash memory does not guarantee that write operation will be performed on erased memory. This situation is shown in Figure 8, in which the function returns fail code (write operation must not be performed), although the memory area designated by parameters is empty.

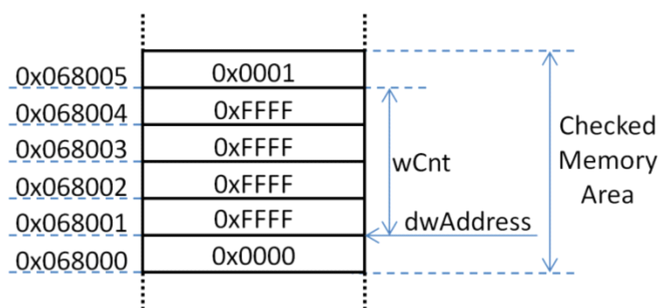

Figure 8. Longword alignment example of FlashDrv_BlankCheck() function

Because of FTFL command preparation and execution time, it is inefficient to use this function for blank check of small memory areas. It might be better to simply read desired memory area and compare its values with `0xFFFFFFFF` constant (remember to use `FlashDrv_Read()` function to access memory above 64 KB with SDM).

Prototype of the function is:

```
UWord8 FlashDrv_BlankCheck(UWord32 dwAddress, UWord16 wCnt);
```

The function performance, parameters, and return codes are listed in following tables.

Table 12. FlashDrv_BlankCheck() function parameters

Parameter Name	Parameter Type	Description
dwAddress	UWord32	Start address in flash (program address space)
wCnt	UWord16	Number of words to check

Table 13. FlashDrv_BlankCheck() function return codes

Return Code	Description
FLASHDRV_SUCCESS	Flash memory area was erased
FLASHDRV_FAIL	Flash memory area is not blank
FLASHDRV_ACCESS_ERROR	Function failed due to invalid parameters (see Table 5)

Table 14. Approximate maximal execution time of FlashDrv_BlankCheck() function

DSC Family	Additional Execution Time	
	Primary Flash	Secondary Flash
MC56F847xx	550 Cycles + 28 µs + 13 µs/KWord	480 Cycles + 26 µs + 26 µs/KWord
MC56F827xx	340 Cycles + 19 µs + 23 µs/KWord	

Example 1 shows `FlashDrv_BlankCheck()` function used for blank-check of part of secondary flash on MC56F847xx DSC family. The `FlashDrv_EraseSector()` function is used to erase memory in case it is not already empty.

Example 1. Use of FlashDrv_BlankCheck()

```
/* Flash Driver Library */
#include "FlashDrv.h"
#include "derivative.h"  /* Peripheral declaration */

int main(void)
{
UWord8 ucResult;

        /* FlashDrv initialization */
        FlashDrv_Init();

        /* Blank check of part of secondary flash memory area
   (in program address space). */
        ucResult = FlashDrv_BlankCheck(0x068001, 4);
        if(ucResult == FLASHDRV_FAIL)
```

```
        {
                /* Memory area is not empty - erase entire sector*/
                ucResult = FlashDrv_EraseSector(0x068001, 4);
        }
return 0;
}
```

4.3.2 FlashDrv_EraseSectors()

This function uses the Erase Flash Sector command (see Section 3.2.3, “Erase flash sector command”). The parameters are word-oriented, so the flash memory area designated by the user can be one word. However, the smallest erasable area has the size of a sector, so every sector covered by function parameters is erased entirely (see the following figure). There is no mechanism that would restore inadvertently erased data.

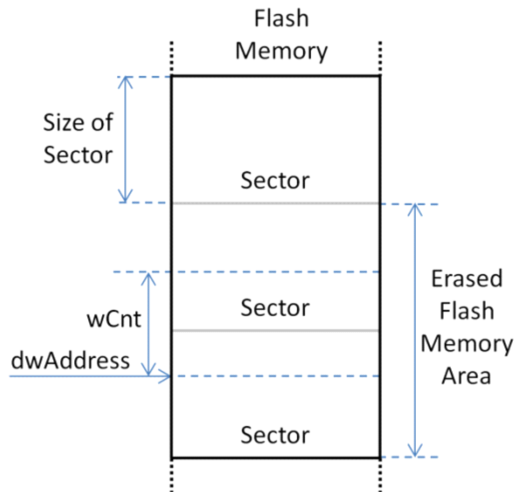

Figure 9. Sector alignment of FlashDrv_EraseSectors() function

Prototype of the function is:

```
UWord8 FlashDrv_EraseSectors(UWord32 dwAddress, UWord16 wCnt);
```

The function performance, parameters, and return codes are listed in the following tables.

Table 15. FlashDrv_EraseSectors() function parameters

Parameter Name	Parameter Type	Description
dwAddress	UWord32	Start address in flash (program address space)
wCnt	UWord16	Number of words to erase

Table 16. FlashDrv_EraseSectors() function return codes

Return Code	Description
FLASHDRV_SUCCESS	Flash memory area was erased successfully
FLASHDRV_FAIL	Verify operation failed
FLASHDRV_ACCESS_ERROR	Function failed due to invalid parameters (see Table 9)
FLASHDRV_PROT_VIOLATION	Protection violation

Table 17. FlashDrv_EraseSectors() execution time

DSC Family	Execution time
MC56F847xx	~6 ms/(erased sector)
MC56F827xx	

Example 2 shows the use of `FlashDrv_EraseSector()` function. The first call erases only the first sector and the second one erases both the first and second sector of secondary flash memory.

Example 2. Use of FlashDrv_EraseSectors() Function

```
/* Flash Driver Library */
#include "FlashDrv.h"
#include "derivative.h"  /* Peripheral declaration */

int main(void)
{
        UWord8  ucResult;
        UWord16 pwData[4] = {1,2,3,4};

        /* Flash Driver Library initialization */
        FlashDrv_Init();

        /* Write data to border of two sectors */
        ucResult = FlashDrv_Write(0x0681FE, pwData, 4);
        // Return code check...

        /* Erase first sector */
        ucResult = FlashDrv_EraseSectors(0x068000, 1);
        // Return code check...
```

```
        /* Erase first two sectors (size of sector is 512W)*/
        ucResult = FlashDrv_EraseSectors(0x068001, 512);
        // Return code check...
        return 0;
}
```

4.3.3 FlashDrv_Write()

The `FlashDrv_Write()` function uses "Program Longword" FTFL flash command. The function has no internal blank-check procedure of memory area that is going to be written. It is advised that the user performs blank-check prior to calling `FlashDrv_Write()` function. Otherwise, there is a danger of overstressing the device. The `FlashDrv_Write()` function parameters are word-oriented. However, the smallest writable area has a size of a longword (four bytes) and the address must be longword-aligned. The function automatically longword-aligns data with empty words if necessary (see Figure 10).

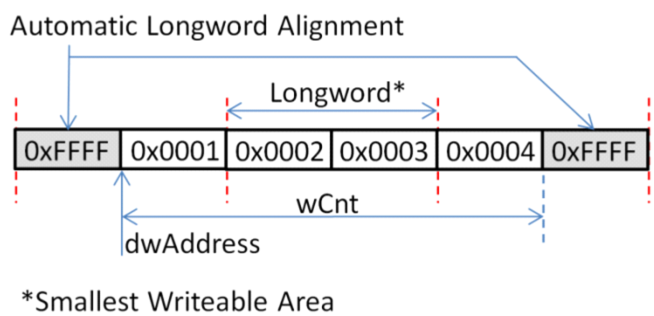

Figure 10. FlashDrv_Write() Function Longword Alignment Example

Prototype of the function is

```
UWord8 FlashDrv_Write(UWord32 dwAddress, const UWord16* pwData, UWord16 wCnt);
```

The function performance, parameters and return codes are listed in following tables.

Table 18. FlashDrv_Write() function parameters

Parameter Name	Parameter Type	Description
dwAddress	UWord32	Start address in flash (program address space)
pwData	UWord16*	Pointer to source data
wCnt	UWord16	Number of words to write

Table 19. FlashDrv_Write() function return codes

Return Code	Description
FLASHDRV_SUCCESS	Function finished successfully
FLASHDRV_FAIL	Verify operation failed
FLASHDRV_ACCESS_ERROR	Function failed due to invalid parameters (see Table 7)
FLASHDRV_PROT_VIOLATION	Protection violation

Table 20. FlashDrv_Write() function approximate execution time

DSC Family	Execution time
MC56F847xx	~35 µs/Longword
MC56F827xx	

Example 3 shows the use of `FlashDrv_Write()` function. The function writes four words into secondary flash on MC56F847xx DSC device.

Example 3. Use of FlashDrv_Write() Function

```
/* Flash Driver Library */
#include "FlashDrv.h"
#include "derivative.h"  /* Peripheral declaration */

int main(void)
{
        UWord8  ucResult;
        UWord16 pwData[4] = {1,2,3,4};

        /* Flash Driver Library initialization */
        FlashDrv_Init();

        /* Write buffer to flash */
        ucResult = FlashDrv_Write(0x068001, pwData, 4);
        // Return code check...

        return 0;
}
```

4.3.4 FlashDrv_Read()

As mentioned in Section 3, "DSC Flash memory description," the `FlashDrv_Read()` function allows to access data over 64 KB, when using Small Data Model (SDM) in C language. The `FlashDrv_Read()` function copies desired number of words from provided flash address to buffer in RAM.

Prototype of the function is

```
asm void FlashDrv_Read(register UWord32 dwAddress,
register UWord16* pwData,
register UWord16 wCnt);
```

The function performance and parameters are listed in the following tables. The input parameters are not checked.

Table 21. FlashDrv_Read() function parameters

Parameter Name	Parameter Type	Description
dwAddress	UWord32	Start address in flash (program address space)
pwData	UWord16*	Pointer to destination data buffer
wCnt	UWord16	Number of words to read

Table 22. FlashDrv_Read() execution time

DSC Family	Execution time
MC56F847xx	30 Cycles + 12 Cycles/Word
MC56F827xx	

Example 4 shows the use of `FlashDrv_Read()` function. The function copies four words from secondary flash memory to buffer in RAM.

Example 4. Use of FlashDrv_Read() Function

```
/* Flash Driver Library */
#include "FlashDrv.h"
#include "derivative.h"  /* Peripheral declaration */

int main(void)
{
        UWord8  ucResult;
        UWord16 pwDataWrite[4] = {1,2,3,4};
UWord16 pwDataRead[4];

        /* Flash Driver Library initialization */
```

```
        FlashDrv_Init();

        /* Write buffer to flash */
        ucResult = FlashDrv_Write(0x068001, pwDataWrite, 4);
        // Return code check...

        /* Copy written data to buffer in RAM */
FlashDrv_Read(0x068001, pwDataRead, 4);

        return 0;
}
```

4.3.5 FlashDrv_Init()

This function initializes FDL. It must be called only once and prior to calling any other FDL function.

As mentioned in Section 3.2, "FTFL commands," it is not possible to simultaneously run code and execute FTFL flash command on the same flash memory block. Therefore, some of the FDL functions must run from RAM. Their copying to RAM is performed by `FlashDrv_Init()` function. However, the user must perform changes in the Linker command file prior to enabling this feature. This is further discussed in Section 5.1, "Linker Command File Modifications."

The second purpose of this function is the initialization of incremental writing feature. The function searches the designated memory area for last written entry and free space for the next one.
The user-defined incremental writing structure variable `FLASHDRV_IWRT_DATA` is initialized with data from the last entry. If no entry was made before, the structure is left initialized with user-defined `FLASHDRV_IWRT_DATA_INIT` data. Also, the incremental writing info structure is initialized; see Example 5 for its definition. The `FLASHDRV_IWRT_INFO` variable is always kept updated by incremental writing functions. Please note that due to internal function of FDL, each entry occupies four extra bytes more than the user defined `FLASHDRV_IWRT_DATA` structure.

Example 5. Incremental Writing Info Structure Definition

```
// User info structure type definition
typedef struct
{
        const UWord32 dwEntryAddr;                  // Address of current entry
        const UWord16 wEntryCnt;                    // Total number of valid entries
        const UWord16 wEntryMaxCnt;                 // Maximum number of entries
        const UWord16 wEntrySize;                   // Size of single entry in words
} FLASHDRV_IWRT_INFO_T, *LPFLASHDRV_IWRT_INFO_T;
```

```
// Incremental writing info structure definition
FLASHDRV_IWRT_INFO_T FLASHDRV_IWRT_INFO;
```

Prototype of the function is:

`void FlashDrv_Init(void);`

The function performance is listed in Table 23.

Table 23. Approximate maximal FlashDrv_Init() execution time

DSC Family	Execution Time	
	Erase All Mode	**Erase Sector Mode**
MC56F847xx	~1440 Cycles + 7 Cycles/Word	~1440Cycles + 7 Cycles/Word + 120 Cycles/Sector
MC56F827xx		

Example 6 shows use of all incremental writing functions. After calling initialization function, which prepares all FDL functions and variables, two new entries are added using `FlashDrv_IncWrite()` function. Each new entry has incremented `dwEntryNum` variable, which was defined as part of `FLASHDRV_IWRT_DATA` structure in `FlashDrv_cfg.h` file (see Section 4.4, “Flash Driver Library configuration”). After that, the `FlashDrv_GetEntry()` is called with information about number of available entries as parameter, to load `FLASHDRV_IWRT_DATA` variable with data from oldest available entry.

Example 6. Use of Incremental Writing Functions Example

```
/* Flash Driver Library */
#include "FlashDrv.h"
#include "derivative.h"  /* Peripheral declaration */

int main(void)
{
        UWord8  ucResult;

        /* FLASHDRV_IWRT_DATA and FLASHDRV_IWRT_INFO variables
         * are initialized with default values*/

        /* Flash Driver Library initialization */
        FlashDrv_Init();

        /* FLASHDRV_IWRT_DATA variable now contains data from newest entry and
* FLASHDRV_IWRT_INFO contains true values. */
```

```
/* Add two new entries. (dwEntryNum variable was defined as part of FLASHDRV_IWRT_DATA_T
 * structure type by user in FlashDrv_cfg.h file) */
         FLASHDRV_IWRT_DATA->dwEntryNum++;
         ucResult = FlashDrv_IncWrite();
// Check return code...
         FLASHDRV_IWRT_DATA->dwEntryNum++;
         ucResult = FlashDrv_IncWrite();
// Check return code...

         /* Load FLASHDRV_IWRT_DATA variable with oldest entry data,
         * FLASHDRV_IWRT_INFO.dwEntryAddr is updated accordingly */
         ucResult = FlashDrv_GetEntry(FLASHDRV_IWRT_INFO.wEntryCnt - 1);
         // Check return code...

         return 0;
}
```

4.3.6 FlashDrv_IncWrite()

This function saves content of user-defined `FLASHDRV_IWRT_DATA` structure variable into next free space in the designated flash memory area. If there is no free flash memory space left, the function erases older entries according to selected mode (see Section 4.1, “Incremental writing feature”). The function can be called repeatedly for adding multiple new entries. The incremental writing user info structure is updated accordingly.

Prototype of the function is

```
UWord8 FlashDrv_IncWrite(void);
```

The function performance and return codes are listed in the following tables.

Table 24. FlashDrv_IncWrite() maximal execution time

DSC Family	Maximal Execution Time	
	Erase All Mode	Erase Sector Mode
MC56F847xx	~6 ms/Sector + 35 µs/Longword	~6 ms + 35 µs/Longword
MC56F827xx		

Table 25. FlashDrv_IncWrite() function parameters

Return Code	Description
FLASHDRV_SUCCESS	Function finished successfully
FLASHDRV_FAIL	Verify operation failed
FLASHDRV_ACCESS_ERROR	Function internal error
FLASHDRV_PROT_VIOLATION	Protection violation

See Example 6 for example of `FlashDrv_IncWrite()` function use.

4.3.7 FlashDrv_GetEntry()

This function can be used to load `FLASHDRV_IWRT_DATA` structure variable with data from older entries. The `wEntryAge` parameter determines which entry to load relatively to newest one. Calling this function with parameter equals to zero loads `FLASHDRV_IWRT_DATA` variable with data from newest entry. The number of available entries can be found in `FLASHDRV_IWRT_INFO` structure variable, as well as other useful information (see Example 5).

Prototype of the function is:

```
UWord8 FlashDrv_GetEntry(UWord16 wEntryAge);
```

The function performance, parameters and return codes are listed in the following tables.

Table 26. FlashDrv_GetEntry() execution time

DSC Family	Execution Time	
	Erase All Mode	Erase Sector Mode
MC56F847xx	50 Cycles + FlashDrv_Read() Call*	280 Cycles + FlashDrv_Read() Call*
MC56F827xx		

Note: * See Table 22. Number of words read by `FlashDrv_Read()` function is equal to size of entry.

Table 27. FlashDrv_GetEntry() Function Parameters

Parameter Name	Parameter Type	Description
wEntryAge	UWord16	Determines which entry to return

Table 28. FlashDrv_GetEntry() Function Return Codes

Return Code	Description
FLASHDRV_SUCCESS	**Function finished successfully**
FLASHDRV_ACCESS_ERROR	**Function failed due to invalid parameter**

See Example 6 for example of `FlashDrv_GetEntry()` function use.

4.3.8 FlashDrv_FMSTRpoll()

This function provides Flash over FreeMASTER functionality. It communicates with PC-side of the FreeMASTER application using runtime modification ability of variables. Based on request, this function calls other FDL functions to perform desired flash memory operations (for more detailed information see Section 4.2, "Flash over FreeMASTER feature"). This function is poll-driven so it must be called repeatedly.

Prototype of the function is:

```
void FlashDrv_FMSTRpoll(void);
```

The execution time of the function depends on the requested operation. Example 7 and Example 8 show examples of `FlashDrv_FMSTRpoll()` function use. Example 7 shows standard peripheral initialization and Example 8 shows peripheral initialization using the Graphic Configuration Tool (GCT). The example requires the FreeMASTER to be set up to communicate via RS232, with enabled ability to modify and read application variables. All necessary peripheral settings are done at the beginning of the example application. Once downloaded to target, the flash programming GUI of FreeMASTER can be used.

Example 7. Use of FlashDrv_FMSTRpoll() function example

```
#include "FlashDrv.h" /* Flash Driver Library */
#include "freemaster.h"  /* FreeMASTER */

int main(void)
{
/* Initialization of peripherals to enable communication with FreeMASTER. It might be
 * necessary to change it depending on used model of DSC.
          * - Enable PLL clock 100 MHz (50 MHz on MC56F827xx)
* - SCI1: baud 9600, 8bit, 1 stop, no parity, RX:ON, TX:ON
* - make GPIOF4(TX1) and GPIOF5(RX1) peripheral driven  */
        OCCS_OSCTL1 = 0x061e;
        OCCS_OSCTL2 = 0xc100;
        OCCS_CTRL = 0x0080U;
        OCCS_CTRL |= 0x0001U;

        COP_CTRL &= ~(0x02U);

        SIM_PCE0 = 0x0002U;
        SIM_PCE1 = 0x0800U;

        GPIOF_PER = 0x0030U;
```

```
        QSCI1_RATE = 0x1458U;
        QSCI1_CTRL1 = 0x000CU;

        /* Initialize FreeMASTER */
        FMSTR_Init();

        /* Flash Driver Library initialization */
        FlashDrv_Init();

        for(;;)
        {
/* FreeMASTER polling function (might be unnecessary, depends on FreeMASTER
 * settings) */
                 FMSTR_Poll();

                 /* Flash over FreeMASTER polling function */
                 FlashDrv_FMSTRpoll();
}
return 0;
}
```

Example 8. Use of FlashDrv_FMSTRpoll() function example with peripheral initialization using graphic configuration tool

```
#include "FlashDrv.h" /* Flash Driver Library */
/* Required DSP56F800E_Quick_Start header */
#include "qs.h"

/* Low-level driver headers for each module used */
#include "intc.h"
#include "gpio.h"
#include "occs.h"
#include "sci.h"
#include "sys.h"
#include "cop.h"
#include "freemaster.h"  /* FreeMASTER */

int main(void)
{
      /* Initialization using Graphic Configuration Tool (GCT).  */
      // Initialize SYS and GPIO modules
```

```
ioctl(SYS, SYS_INIT, NULL);
ioctl(COP, COP_INIT, NULL);

ioctl(GPIO, GPIO_INIT_ALL, NULL);

// Initialize UART
ioctl(SCI_RS232, SCI_INIT, NULL);
/* Initialize FreeMASTER */
FMSTR_Init();

/* Flash Driver Library initialization */
FlashDrv_Init();

for(;;)
{
      /* FreeMASTER polling function (might be unnecessary, depends on FreeMASTER
       * settings) */
       FMSTR_Poll();

       /* Flash over FreeMASTER polling function */
       FlashDrv_FMSTRpoll();
}
return 0;
        }
```

4.4 Flash Driver Library configuration

All Flash Driver Library settings are done in `FlashDrv_cfg.h` file. This file contains various preprocessor definitions:

- `FLASHDRV_FLSHCNT` – This line defines the number of flash memory blocks. This could be either one for MC56F827xx DSC family or two for MC56F847xx DSC family. This information must always be set correctly, as the DSC family is determined based on this definition.
- `FLASHDRV_PRIMARY_START` – Start address of primary flash in program address space.
- `FLASHDRV_PRIMARY_END` – End address of primary flash in program address space.
- `FLASHDRV_PRIMARY_SECTOR_SIZE` – Size of sector in primary flash memory.
- `FLASHDRV_SECONDARY_START` – Start address of secondary flash in program address space.
- `FLASHDRV_SECONDARY_END` – End address of secondary flash in program address space.
- `FLASHDRV_SECONDARY_SECTOR_SIZE` – Size of sector in secondary flash memory.

All previous address and sector size definitions can be found in reference manuals of used DSC family.

- `FLASHDRV_COPY2RAM` – The non-zero value enables copying of internal flash command function into RAM. This option could be disabled when using secondary flash memory only on MC56F847xx DSC family.
- `FLASHDRV_IWRT_ENABLE` – This option enables incremental writing feature. The `FlashDrv_IncWrite()` and `FlashDrv_GetEntry()` functions are available only when this definition has non-zero value.

- FLASHDRV_IWRT_START – Starting address of designated incremental writing flash memory area. This address must be sector aligned. Both primary and secondary flash memory blocks can be used.
- FLASHDRV_IWRT_SECT_CNT – This option defines number of sectors designated for incremental writing. This definition together with FLASHDRV_IWRT_START determine size of designated incremental writing flash memory area.
- FLASHDRV_IWRT_ERASE_ALL – This option selects between "Erase All" and "Erase Sector" mode (see Section 4.1, "Incremental writing feature"). When zero value is asserted, "Erase Sector" mode is selected.
- FLASHDRV_FMSTR_ENABLE - The non-zero value enables Flash over FreeMASTER feature.
- FLASHDRV_FMSTR_BUFF_SIZE - Size of I/O data buffer in bytes. This value must not be lower than the one selected in FreeMASTER's Flash Programming window.

Example of incremental writing data structure definition is shown in Example 9. Here, the user should list all variables that he wants to store using incremental writing feature. The size of single structure variable should not exceed the size of sector.

Example 9. Example of incremental writing data structure type definition

```
typedef struct
{
        /* List of variables that will be backed-up using incremental writing feature
         * For example:*/
        unsigned char ucString[13];
        signed int wNumber;
} FLASHDRV_IWRT_DATA_T, *LPFLASHDRV_IWRT_DATA_T;
```

FDL creates only one instance of FLASHDRV_IWRT_DATA_T data type, named FLASHDRV_IWRT_DATA. The user has option to define default value of this global variable, using FLASHDRV_IWRT_DATA_INIT definition. Example 9 can be initialized with line

```
#define FLASHDRV_IWRT_DATA_INIT "Hello World!", 0x1234
```

5 Installation guide

Flash Driver Library consists of four files. The FlashDrv_cfg.h file was described in Section 4.4, "Flash Driver Library configuration." Remaining FlashDrv.h, FlashDrv_56F8xxx.h and FlashDrv.c form the core of Flash Driver Library.

The following step-by-step guide shows how to create a new project and include Flash Driver Library in CodeWarrior v10.x. Linker Command File modifications, which are necessary to enable FLASHDRV_COPY2RAM feature, are described in Section 5.1, "Linker Command File Modifications."

1. Open the CodeWarrior for Microcontrollers v10.x IDE.
2. Select File > New > Bareboard Project.
3. The New Bareboard Project dialog box appears.

4. Type a name for the project you want to create in the Project name text box. For example, “FDL test project” and select location to store the new project.
5. Click Next button to continue the action.
6. Select the required device from the list of available devices. For example, expand 56800/E (DSC) > MC56F847xx and select MC56F84789 device as target processor from the list.
7. Click Next button to continue the action.
8. Select connection type and click Finish button to finish the action.
9. The New Bareboard Project dialog box closes.
10. The wizard creates a new project according to your specifications. You can access the project from the CodeWarrior Projects view in the opened Workbench window
11. Right-click on project name in CodeWarrior Projects view and select New > Folder. Type “FlashDrv” as folder name in newly opened New Folder dialog box and click Finish.
12. Right-click on “FlashDrv” folder and select Add Files... Find and select Flash Driver Library files (`FlashDrv_cfg.h`, `FlashDrv.h` and `FlashDrv.c`). Click Open to finish the action.
13. Now once again right-click on project name in CodeWarrior Projects view and click Properties. In newly opened window select C/C++ General > Paths and Symbols.
14. In the Include bookmark select “c,C,cc,cxx,cpp” and click Add button. Now write “${ProjDirPath}/FlashDrv” into Add directory path window and click OK and then Apply.
15. Double click on Sources folder in CodeWarrior Projects view and open main.c file.
16. Add line “`#include "FlashDrv.h"`” at the beginning of the main.c file, where other inclusions are.
17. Perform all necessary changes in `FlashDrv_cfg.h` file as described in Section 4.4, “Flash Driver Library configuration.”

5.1 Linker Command File Modifications

The CodeWarrior Executable and Linking Format (ELF) Linker makes a program file out of the object files of your project. The linker also allows manipulation of code in different ways (define variable during linking, change alignment…). All of these functions are accessed through commands in the linker command file (LCF). The linker command file has its own language with keywords, directives, and expressions. For more information see the CodeWarrior Build Tools reference manual for 56800/E Digital Signal Controllers, available at freescale.com.

Linker command files contain three main segments:

- Memory Segment
- Closure Blocks (optional)
- Sections Segment

In the memory segment, the available memory is divided into segments. The incomplete example of memory segment on MC56F84789 device is in Example 10. Each segment typically contains name, access permission (RWX mean Read Write and eXecute access), start address of memory segment (starting with ORIGIN keyword) and size of memory segment (LENGTH keyword). Memory segments with RWX (RX) attributes are placed into program memory while RW attributes are placed into data memory. Memory segment, which is used to store FDL functions in RAM is called `.p_ram_RAM_code`. It is created by default by CodeWarrior New Project Wizard on both supported DSC families (comment out the line if necessary). The segment must have size at least of 120 words.

Example 10. Example of Memory Segment in Linker Command File

```
MEMORY {
       # Program Memory space
         .p_interruptsboot_ROM (RX): ORIGIN = 0x000000, LENGTH = 0x000004
.p_interrupts_ROM     (RX): ORIGIN = 0x000004, LENGTH = 0x000200
         .p_flash_ROM          (RX): ORIGIN = 0x000208, LENGTH = 0x01FDF8
         .p_ram_RAM_code       (RX): ORIGIN = 0x060000, LENGTH = 0x002000
         .p_boot_ROM           (RX): ORIGIN = 0x068000, LENGTH = 0x004000
         # Additional sections...

# Data Memory space
         .x_internal_RAM_code   (RW): ORIGIN = 0x000000, LENGTH = 0x002000
         .x_internal_RAM_data   (RW): ORIGIN = 0x002000, LENGTH = 0x002000
         .x_platform            (RW): ORIGIN = 0x00C000, LENGTH = 0x002000
         # Additional sections...
}
```

Inside the sections segment, the contents of memory segments, and any global symbols to be used in the output file are defined. Each section typically consist of name, start address of section (optional; starting with keyword AT) and content (`.text` extension for code). It is possible to define variables inside the sections. Variables with name that starts with letter “F” are global and it is possible to access them in the rest of the code. A shortened example of typical sections segment generated by CodeWarrior New Project Wizard, including changes necessary to run FDL functions from RAM can be found in Example 11. Code added by the user is highlighted. First, the end address `F_FDL_START` of `.ApplicationCode` section is stored. The `.FDL_RAM` section is placed in flash memory behind this section using keyword AT. Body of `.FDL_RAM` section stores content and calculates all necessary addresses.

Example 11. Example of Sections Segment in Linker Command File

```
SECTIONS {
         # Additional sections...
.ApplicationCode :
{
# Section content...
. = ALIGN(2);
# Starting address of Flash Driver Library code in flash
```

```
                  F_FDL_START = .;
} > .p_flash_ROM



# Flash Driver Library RAM code section
         .FDL_RAM : AT(F_FDL_START)
         {
         _xRAM_CODE_START = .;
         * (p_FDL_RAM_CODE.text)
. = ALIGN(2);
_xRAM_CODE_END = .;

                  F_CODE_SIZE         = _xRAM_CODE_END - _xRAM_CODE_START;
                  F_XRAM_CODE_START = _xRAM_CODE_START - 0x060000; # 0x00F000 for MC56F827xx
                  x_FDL_START = F_FDL_START + 0x020000;                # 0x004000 for
MC56F827xx
                  F_PROM_CODE_STARTH = (x_FDL_START / 65536) & 0xFFFF;
                  F_PROM_CODE_STARTL = x_FDL_START & 0xFFFF;

                  F__pROM_data_start = F_FDL_START + F_CODE_SIZE;
} > .p_ram_RAM_code

.data_in_p_flash_ROM : AT(F__pROM_data_start)
{
# Section content...
} > .p_flash_ROM_data
# Additional sections...
}
```

6 Summary

In summary, the necessary steps for a newly-created project to enable FDL functions running from RAM are:

1. In CodeWarrior Projects view select your project and go to Project_Settings > Linker_Files and open proper `*.cmd` file.
2. Go to memory segment and uncomment line that defines `.p_ram_RAM_code` memory segment (as shown in Example 10).

3. Go to sections segment and find `.ApplicationCode` section. Go to its end and replace "`__pROM_data_start = .;`" line with "`F_FDL_START = .;`" line (highlighted in Example 11). Replace all the occurrences of `__pROM_data_start`" in remaining code with "`F__pROM_data_start`."
4. Now insert whole `.FDL_RAM` section code (highlighted in Example 11) behind the `.ApplicationCode` section (before `.data_in_p_flash_ROM` section).
5. On MC56F827xx family the user must change two constants inside `.FDL_RAM` section body, as indicated by comments in Example 11.
6. Go to `FlashDrv_cfg.h` file and set `FLASHDRV_COPY2RAM` definition to non-zero value.

7 Revision history

Table 29. Revision history

Revision number	Date	Substantial changes
0	12/2013	Initial release
1	11/2014	Applied several small updates, upgraded examples

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4860
Rev. 1
11/2014