The following document contains information on Cypress products. The document has the series name, product name, and ordering part numbering with the prefix “MB”. However, Cypress will offer these products to new and existing customers with the series name, product name, and ordering part number with the prefix “CY”.

How to Check the Ordering Part Number

1. Go to www.cypress.com/pcn.
2. Enter the keyword (for example, ordering part number) in the **SEARCH PCNS** field and click **Apply**.
3. Click the corresponding title from the search results.
4. Download the Affected Parts List file, which has details of all changes

For More Information

Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress

Cypress is the leader in advanced embedded system solutions for the world's most innovative automotive, industrial, smart home appliances, consumer electronics and medical products. Cypress' microcontrollers, analog ICs, wireless and USB-based connectivity solutions and reliable, high-performance memories help engineers design differentiated products and get them to market first. Cypress is committed to providing customers with the best support and development resources on the planet enabling them to disrupt markets by creating new product categories in record time. To learn more, go to www.cypress.com.

AN204878

FM3 MB9A/BFXXX With Flash Programming

This application note describes how to program the embedded Flash memory while the application is running. It also shows how to adjust the IAR and KEIL compilers for generating RAM code.

Contents

1 Introduction....1
2 Programming Principle....1
2.1 Programming via RAM code (Main Flash)....1
2.2 Programming the Work Flash....2
2.3 Main Flash Memory Organization....3
2.4 Work Flash Memory Organization....5
3 Flash Programming Sequences and Registers....5
3.1 Flash Interface....5
3.2 Main Flash Access Size....8
3.3 Work Flash Access Size....9
4 Flash Programming Software Example....9
4.1 Main Flash Sector Erase – Type 0 and 2 Devices....9
4.2 Main Flash Programming – Type 0 Devices....12
4.3 Type 1 Devices....14
4.4 Main Flash Programming – Type 2 Devices....14
4.5 Project Adjustments for generating RAM Code and automatically Copying at Start-Up Phase....16
4.6 Intercompatibility for different Compilers....19
5 Document History....21
Worldwide Sales and Design Support....22

1 Introduction

This application note describes how to program the embedded Flash memory while the application is running. It also shows how to adjust the IAR and KEIL compilers for generating RAM code.

2 Programming Principle

This Chapter shows the Principle of Flash Programming

2.1 Programming via RAM code (Main Flash)

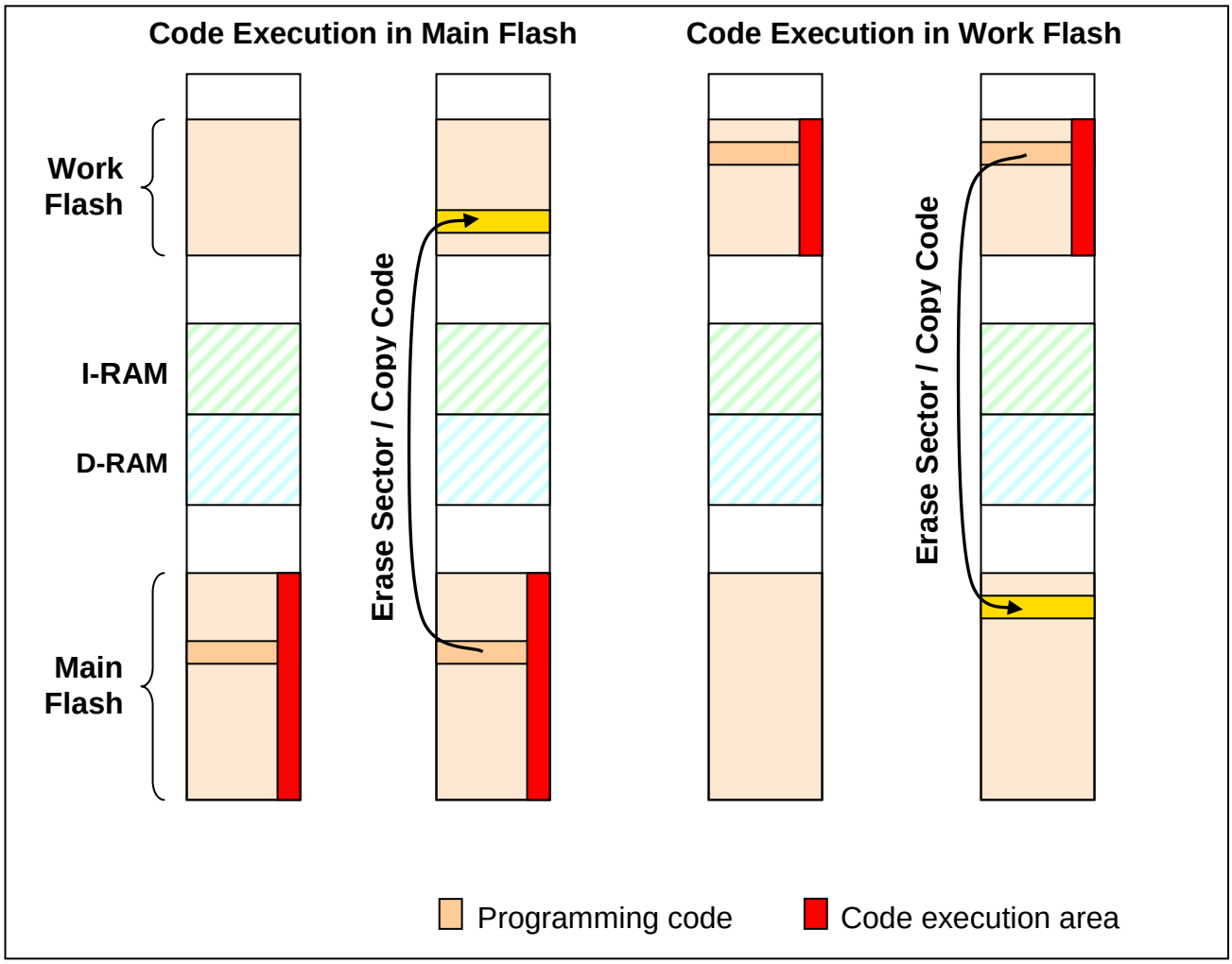
Because for Flash programming the Flash interface has to be set to the command sequencer mode, it cannot be used for reading. This means that the CPU is not able to fetch any instruction from it in this mode.

Therefore for any usage of the automatic programming algorithm with its command sequences the code execution must be done outside the Flash memory. This can be done in external memory, but more useful in the Instruction-RAM area of the FM3 starting from address 0x2000.0000.

The user has to take care, that the programming code itself has to be copied from (constant) ROM area to the I-RAM area before executing it (Step1). Normally this can be done by compiler and linker settings in the used project builder IDE, which is explained later.

The following graphic illustrates the mechanism and principle

Figure 1. Main Flash programming principle



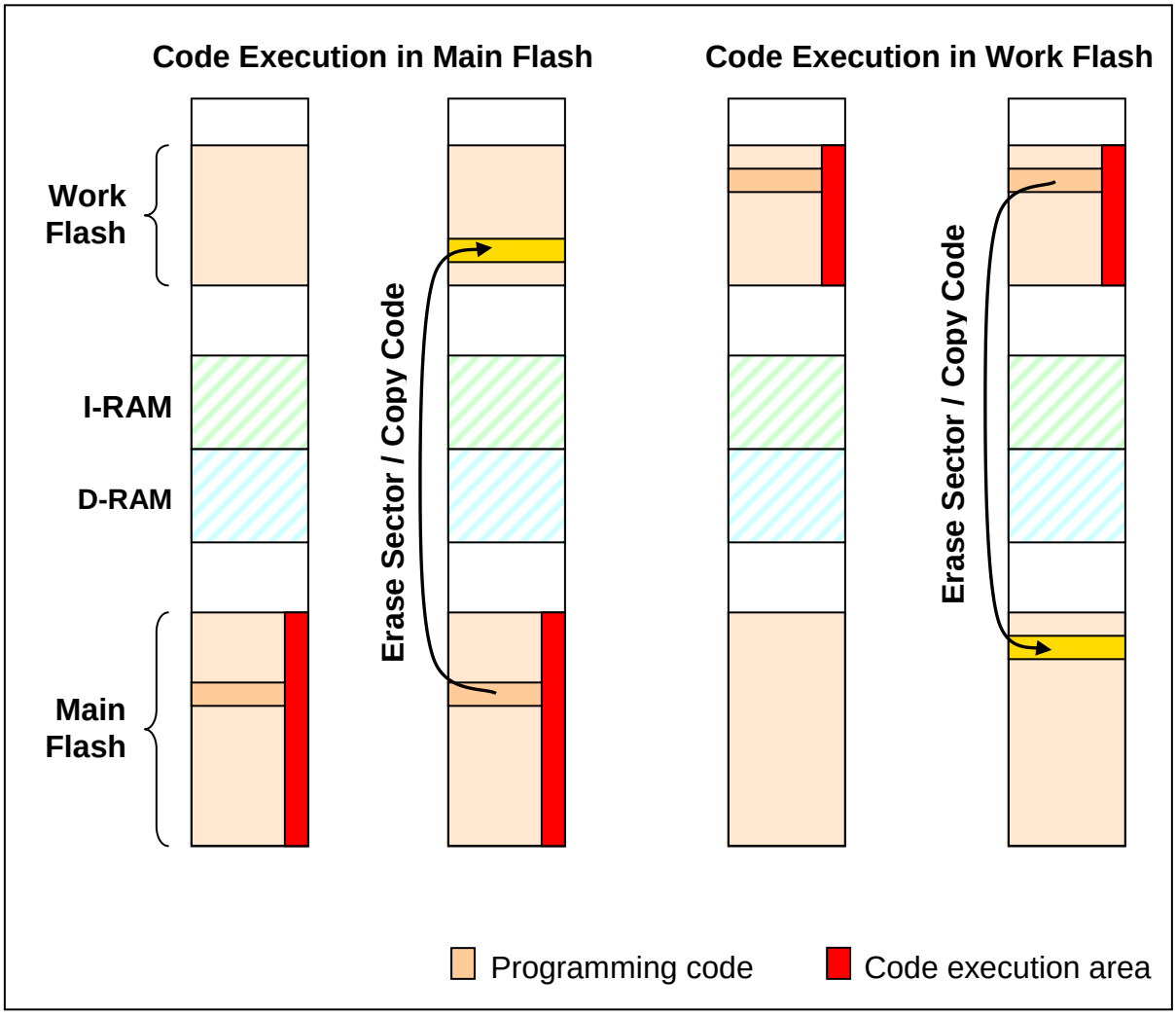
After copying the code in Step 1, the application has to jump to the copied RAM code. Here it is allowed to set the Flash memory to erase/programming mode. In the step 2 in illustration above a certain sector is erased and programmed. After successful Flash content change the application can jump back to the Flash area for normal code execution (Step 3).

2.2 Programming the Work Flash

If a device supports Work Flash no RAM code is needed for programming. The Work Flash is a second independent Flash memory, which can be erased and programmed from the Main Flash. This is also possible vice versa.

The following graphic illustrates this:

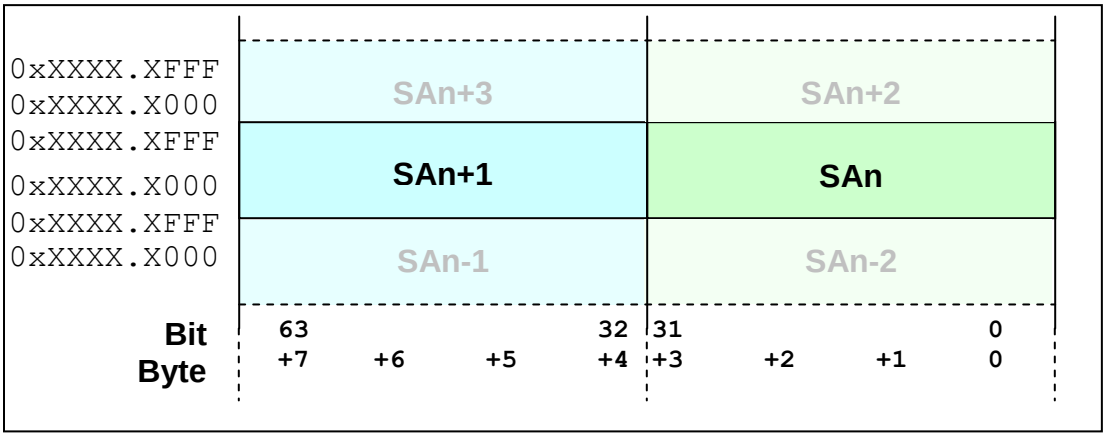
Figure 2. : Work Flash programming principle



2.3 Main Flash Memory Organization

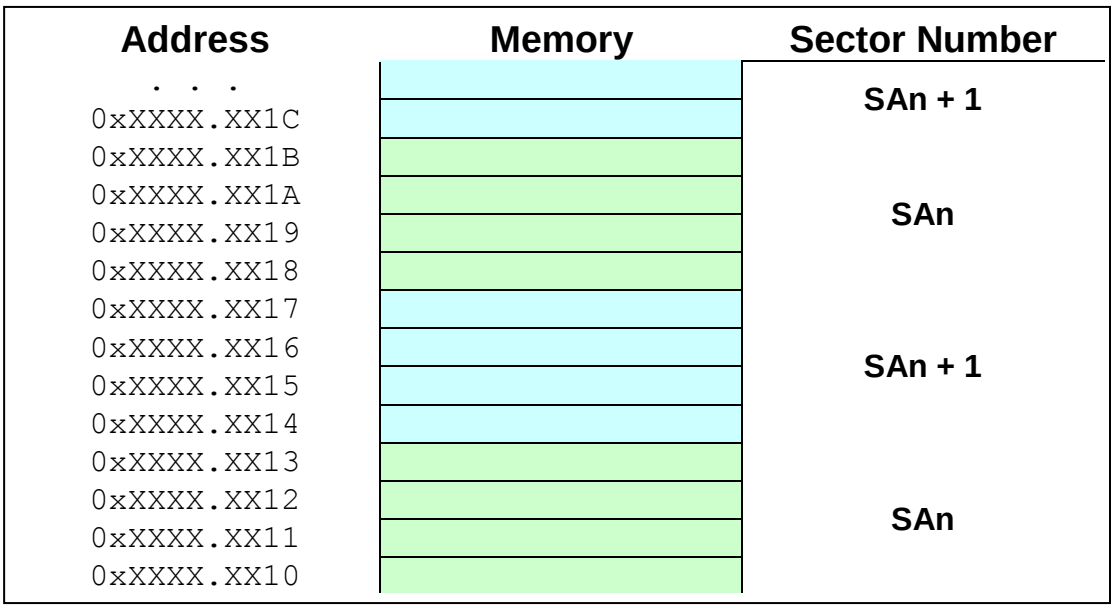
Note that the topology of the Flash sectors is organized by lower- and upper-32-bit word sectors, which results in a 64-bit-wide Flash memory. The following graphic shows this topology:

Figure 3. Main Flash Sector Topology



This sector topology results in the following sector/memory address organization within two upper- and lower-32-bit-word sectors (SAn and SAn+1):

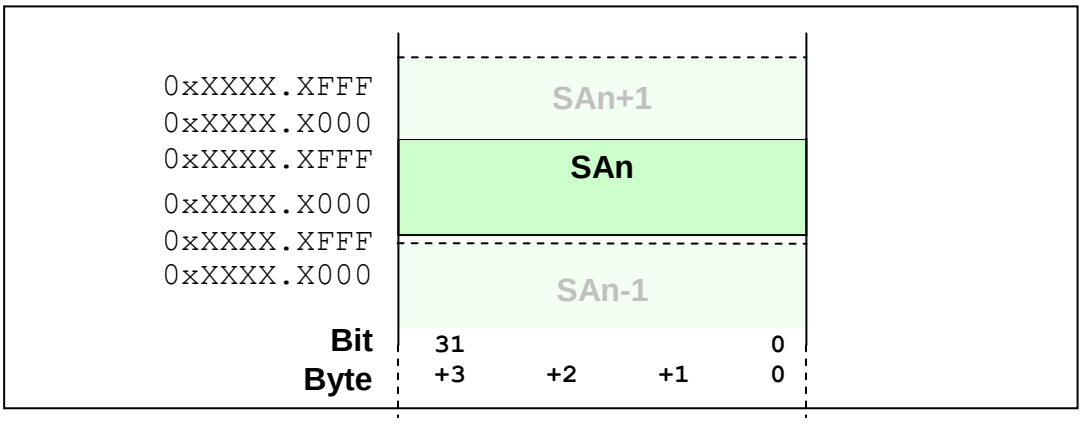
Figure 4. Main Flash Memory Organization



2.4 Work Flash Memory Organization

The Work Flash is organized in consecutive 32-Bit Words without interlace:

Figure 5. Work Flash Sector Topology



3 Flash Programming Sequences and Registers

This Chapter Shows the Flash Sequences and Relevant Registers

3.1 Flash Interface

The FM3 embedded Flash memory provides a Flash interface with an automatic programming algorithm. For unlocking this algorithm certain address/16-bit-data sequences have to be written to the Flash area (where the upper 8 data bits are “don’t care”).

Any write access to the Flash memory area triggers the sequencer, but only certain addresses with certain data unlock a Flash memory command

3.1.1 Main Flash (Chip) Erase Command Sequence

The Main Flash (chip) erase command sequence consists of the following address/data write accesses:

Table 1. Main Flash (Chip) Erase Command Sequence

Address Device Type 0 and 2	Address Device Type 1	Data	Comment
0x0000[1].1550	0x0000[1].0AA8	0xXXAA	1st sequence write
0x0000[1].0AA8	0x0000[1].0554	0xXX55	2nd sequence write
0x0000[1].1550	0x0000[1].0AA8	0xXX80	3rd sequence write
0x0000[1].1550	0x0000[1].0AA8	0xXXAA	4th sequence write
0x0000[1].0AA8	0x0000[1].0554	0xXX55	5th sequence write
0x0000[1].1550	0x0000[1].0AA8	0xXX10	6th sequence write

Note: This sequence only erases the whole Flash, if the device does not support a Work Flash area. If Work Flash is provided, it has to be erased separately (see below).

3.1.2 Main Flash Sector Erase Command Sequence

The Main Flash sector erase command sequence consist of the following address/data write accesses:

Table 2. Main Flash Sector Erase Command Sequence

Address Device Type 0 and 2	Address Device Type 1	Data	Comment
0x00001.1550	0x00001.0AA8	0xXXAA	1st sequence write
0x00001.0AA8	0x00001.0554	0xXX55	2nd sequence write
0x00001.1550	0x00001.0AA8	0xXX80	3rd sequence write
0x00001.1550	0x00001.0AA8	0xXXAA	4th sequence write
0x00001.0AA8	0x00001.0554	0xXX55	5th sequence write
SAn	SAn	0xXX30	6th sequence write with SAn address within sector to be erased

Note: The sequence addresses for the unlock of the erase command can be any address within the (Main) Flash area. Because the (Main) Flash memory starts from address 0x0000.0000 in the FM3 architecture, the upper 16-bit can be left as 0x0000.

3.1.3 Main Flash 16-/32-Bit Word Write Command Sequence

For writing a 16-bit word to an erased Flash cell, the following command sequence has to be used:

Table 3. Main Flash Write Data Command Sequence

Address Device Type 0 and 2	Address Device Type 1	Data	Comment
0x00001.1550	0x00001.0AA8	0xXXAA	1st sequence write
0x00001.0AA8	0x00001.0554	0xXX55	2nd sequence write
0x00001.1550	0x00001.0AA8	0xXXA0	3rd sequence write
PAddr	PAddr	PData	4th actual write access to PAddr and PData. The address at PAddr will contain PData after successful programming.

For writing a 32-Bit word including automatic ECC calculation (Type 2 devices) program the 1st lower 16-Bit word to the desired Flash address and the 2nd upper 16-Bit word the Flash address + 2 afterwards. The second programming step programs also the ECC cells.

Note: That the sequence addresses for the unlock of the erase command can be any address within the (Main) Flash area. Because the (Main) Flash memory starts from address 0x0000.0000 in the FM3 architecture, the upper 16-bit can be left as 0x0000.

3.1.4 Work Flash Erase Command Sequence

The Work Flash erase command sequence consists of the following address/data write accesses:

Table 4. Work Flash Erase Command Sequence

Address	Data	Comment
0x200C2.0AA8	0xXXAA	1st sequence write
0x200C2.0554	0xXX55	2nd sequence write
0x200C2.0AA8	0xXX80	3rd sequence write
0x200C2.0AA8	0xXXAA	4th sequence write
0x200C2.0554	0xXX55	5th sequence write
0x200C2.0AA8	0xXX10	6th sequence write

Note: That the sequence addresses for the unlock of the Work Flash erase command can be any address within the Work Flash area. Assume the Work Flash memory starts from address 0x200C.0000, the upper 16-bit then have to be 0x200C.

3.1.5 Work Flash Sector Erase Command Sequence

The Work Flash sector erase command sequence consists of the following address/data write accesses:

Table 5. Work Flash Sector Erase Command Sequence

Address	Data	Comment
0x200C^2.0AA8	0xXXAA	1st sequence write
0x200C^2.0554	0xXX55	2nd sequence write
0x200C^2.0AA8	0xXX80	3rd sequence write
0x200C^2.0AA8	0xXXAA	4th sequence write
0x200C^2.0554	0xXX55	5th sequence write
SAn	0xXX30	6th sequence write with SAn address within sector to be erased

3.1.6 Main Flash 16-/32-Bit Word Write Command Sequence

For writing a 16-bit word to an erased Flash cell, the following command sequence has to be used:

Table 6. Work Flash Write Data Command Sequence

Address	Data	Comment
0x200C^2.0AA8	0xXXAA	1st sequence write
0x200C^2.0554	0xXX55	2nd sequence write
0x200C^2.0AA8	0xXXA0	3rd sequence write
PAddr	PData	4th actual write access to PAddr and PData. The address at PAddr will contain PData after successful programming.

Note: That the sequence addresses for the unlock of the Work Flash commands can be any address within the Work Flash area. Assume the Work Flash memory starts from address 0x200C.0000, the upper 16-bit then have to be 0x200C.

3.1.7 Read/Reset

This command can be used to abort any Flash command and reset it to the default read state. Be careful when using it together with chip or sector erase. Incomplete erase states may result.

The read/reset command is issued when writing a 0xXXF0 to any of the Flash memory addresses.

3.1.8 Sector Erase Suspend and Restart

When starting a sector erase by its command sequence, it can be halted by the suspend command. To issue a suspend command, write 0xXXB0 to any of the sector's addresses.

When the sector erase is suspended and in the halt state, it is allowed to write new data to another sector. This is useful, if urgent data have to be written to a non-volatile memory area, but a sector erase was started.

After any write command to another sector, the erase can be resumed by writing 0xXX30 to any on the sector's addresses.

3.1.9 Automatic Programming Algorithm Run States

By reading any Flash address after issuing a command sequence, the lower 8 bits correspond to certain state flags of the Flash interface.

Table 7. Flags of Automatic Programming Algorithm State

Bit Number	DQ7	DQ6	DQ5	DQ4	DQ3	DQ2	DQ1	DQ0
Name	DPOL	TOGG	TLOV	-	SETI	TOGG2	-	-

After any command a small user code should check these bits by polling to determine if the command was finished successful or a time-out has occurred.

There a two different ways for this: Data polling and data toggle algorithm.

The most important bits are DPOL, TLOV and SETI, if the data polling algorithm is used. This method is described in this application note. Refer to the Flash programming manual for details of the data toggle algorithm.

Another method is to use the Flash Status Register FSTR.

Bit Number	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	EER1	HNG	RDY

The RDY bit shows the current state of erase or program. RDY == 1 shows a command finished state.

3.2 Main Flash Access Size

Special attention has to be paid to the Flash Access Size Register (FASZR). For normal code and data fetch the Flash interface is 32-bit wide, but for accessing the Flash interface by write sequence and flag polling it has to be set to 16-bit data width.

The following table shows the two different allowed settings.

Table 8. ASZ Bit Configuration of Main Flash Access Size Register (FASZR)

Bit Number	7	6	5	4	3	2	1	0
Name	reserved						ASZ	
Not allowed	0	0	0	0	0	0	0	0
16-Bit read/write (Erase/Program)	0	0	0	0	0	0	0	1
32-Bit read (ROM mode)	0	0	0	0	0	0	1	0
Not allowed	0	0	0	0	0	0	1	1
Not allowed	0	0	0	0	0	1	0	0
	1	1	1	1	1	1	1	1

Note: This register shall only be written, when code is not executed in the Flash area!

3.3 Work Flash Access Size

Special attention has to be paid to the Work Flash Access Size Register (WFASZR), if Work Flash is provided. For normal code and data fetch the Flash interface is 32-bit wide, but for accessing the Flash interface by write sequence and flag polling it has to be set to 16-bit data width.

1 Only available at ECC Flash support.

The following table shows the two different allowed settings.

Table 9. ASZ Bit Configuration of Work Flash Access Size Register (WFASZR)

Bit Number	7	6	5	4	3	2	1	0
Name	reserved							ASZ
16-Bit read/write (Erase/Program)	0	0	0	0	0	0	0	0
32-Bit read (ROM mode)	0	0	0	0	0	0	0	1
Not allowed	0	0	0	0	0	0	1	1
	0	0	0	0	0	1	0	0
	1	1	1	1	1	1	1	1

Note: This register shall only be written, when code is not executed in the Flash area!

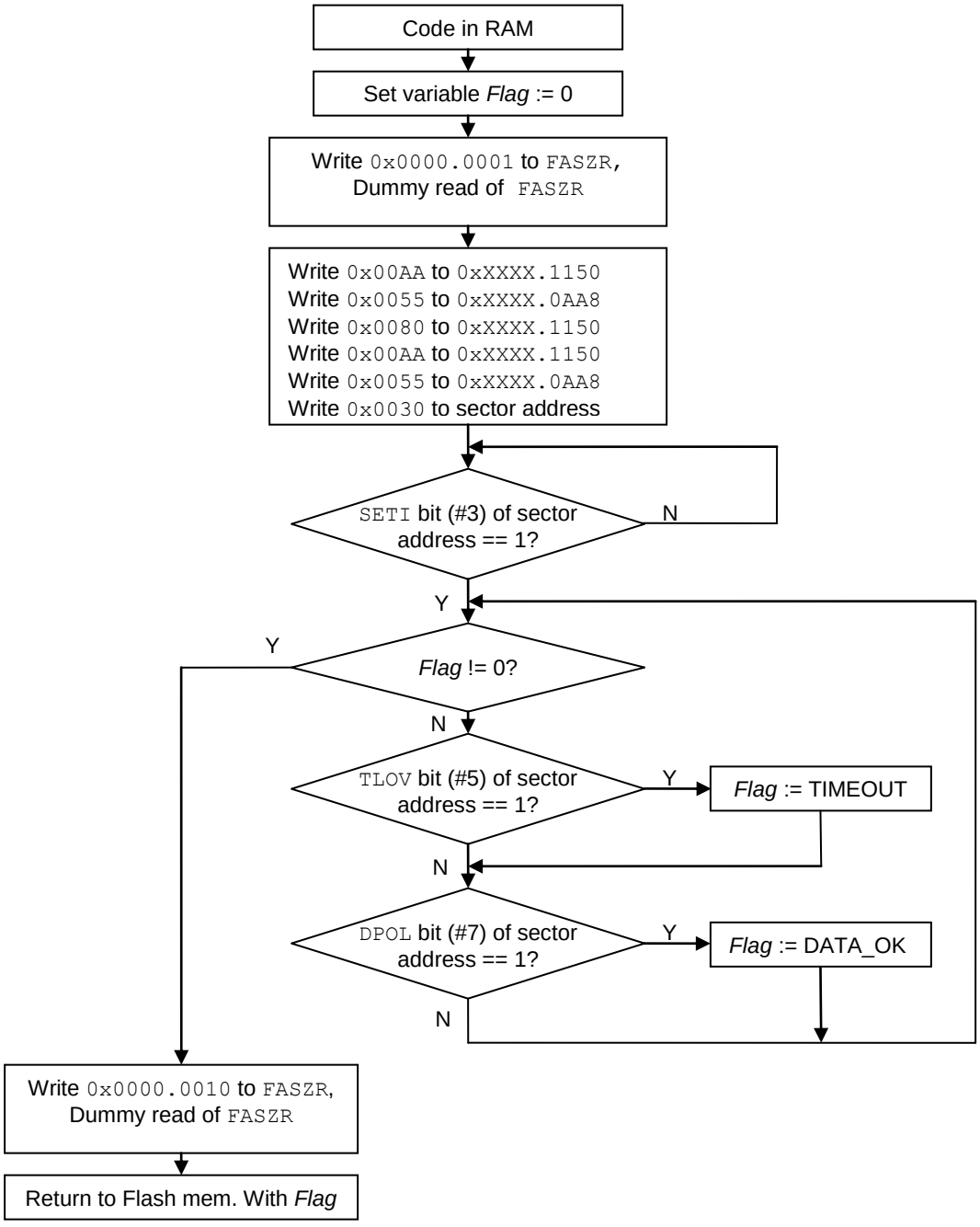
4 Flash Programming Software Example

How to perform a sector erase and how to program new data.

4.1 Main Flash Sector Erase – Type 0 and 2 Devices

The following flowchart shows how to program a RAM code section for performing a Flash sector erase.

Figure 6. Flash Sector Erase Type 0 and 2 Devices Flow Chart



Note: With Work Flash sequence addresses this flow diagram is also valid for Work Flash except that its execution does not need to be in RAM area.

The program code of the sector erase RAM function may look like the following example:

Figure 7. Flash Sector Erase Type 0 and 2 Devices Software Example

```
#include "mb9bfxxx.h"

#define FLASH_SEQ_1550 ((volatile uint16_t*) 0x00001550) // sequence address 1
#define FLASH_SEQ_0AA8 ((volatile uint16_t*) 0x00000AA8) // sequence address 2

#define FLASH_SECTOR_ERASE_1 0x00AA   // sector erase commands
#define FLASH_SECTOR_ERASE_2 0x0055
#define FLASH_SECTOR_ERASE_3 0x0080
#define FLASH_SECTOR_ERASE_4 0x00AA
#define FLASH_SECTOR_ERASE_5 0x0055
#define FLASH_SECTOR_ERASE_6 0x0030

#define FLASH_DQ7 0x0080     // data polling flag bit (DPOL) position
#define FLASH_DQ5 0x0020     // time limit exceeding flag bit (TLOV) position
#define FLASH_DQ3 0x0008     // sector erase timer flag bit (SETI) position

#define FLASH_TIMEOUT_ERROR -1
#define FLASH_OK            1

#ifdef __ICCARM__
  #pragma section = ".flash_ram_code"
#endif

#ifdef __ICCARM__
  __ramfunc
#elif  __CC_ARM
  __attribute__ ((section (".ramfunc")))
#else
  #error please check compiler and linker settings for RAM code
#endif
Int32_t FlashRomEraseSector(uint32_t u32SectorEraseAddress)
{
  Volatile int32_t i32FlashFlag = 0;
  Volatile uint32_t u32DummyRead;

  FM3_FLASH_IF->FASZR &= 0xFFFD;      // ASZ[1:0] = 2'b01
  FM3_FLASH_IF->FASZR |= 1;
  u32DummyRead = FM3_FLASH_IF->FASZR; // dummy read of FASZR

  *(FLASH_SEQ_1550) = FLASH_SECTOR_ERASE_1;
  *(FLASH_SEQ_0AA8) = FLASH_SECTOR_ERASE_2;
  *(FLASH_SEQ_1550) = FLASH_SECTOR_ERASE_3;
  *(FLASH_SEQ_1550) = FLASH_SECTOR_ERASE_4;
  *(FLASH_SEQ_0AA8) = FLASH_SECTOR_ERASE_5;
  *(volatile uint32_t*) u32SectorEraseAddress = FLASH_SECTOR_ERASE_6;

  // sector erase timer ready?
  While ((*(volatile uint16_t *) u32SectorEraseAddress & FLASH_DQ3) != FLASH_DQ3);

  While (0 == i32FlashFlag)
  {
    // Flash timeout?
    If ((*(volatile uint16_t *) u32SectorEraseAddress & FLASH_DQ5) == FLASH_DQ5)
    {
      i32FlashFlag = FLASH_TIMEOUT_ERROR;
    }

    // Data correct?
    if((*(volatile uint16_t *)u32SectorEraseAddress & FLASH_DQ7) == FLASH_DQ7)
    {
      i32FlashFlag = FLASH_OK;
    }
  }

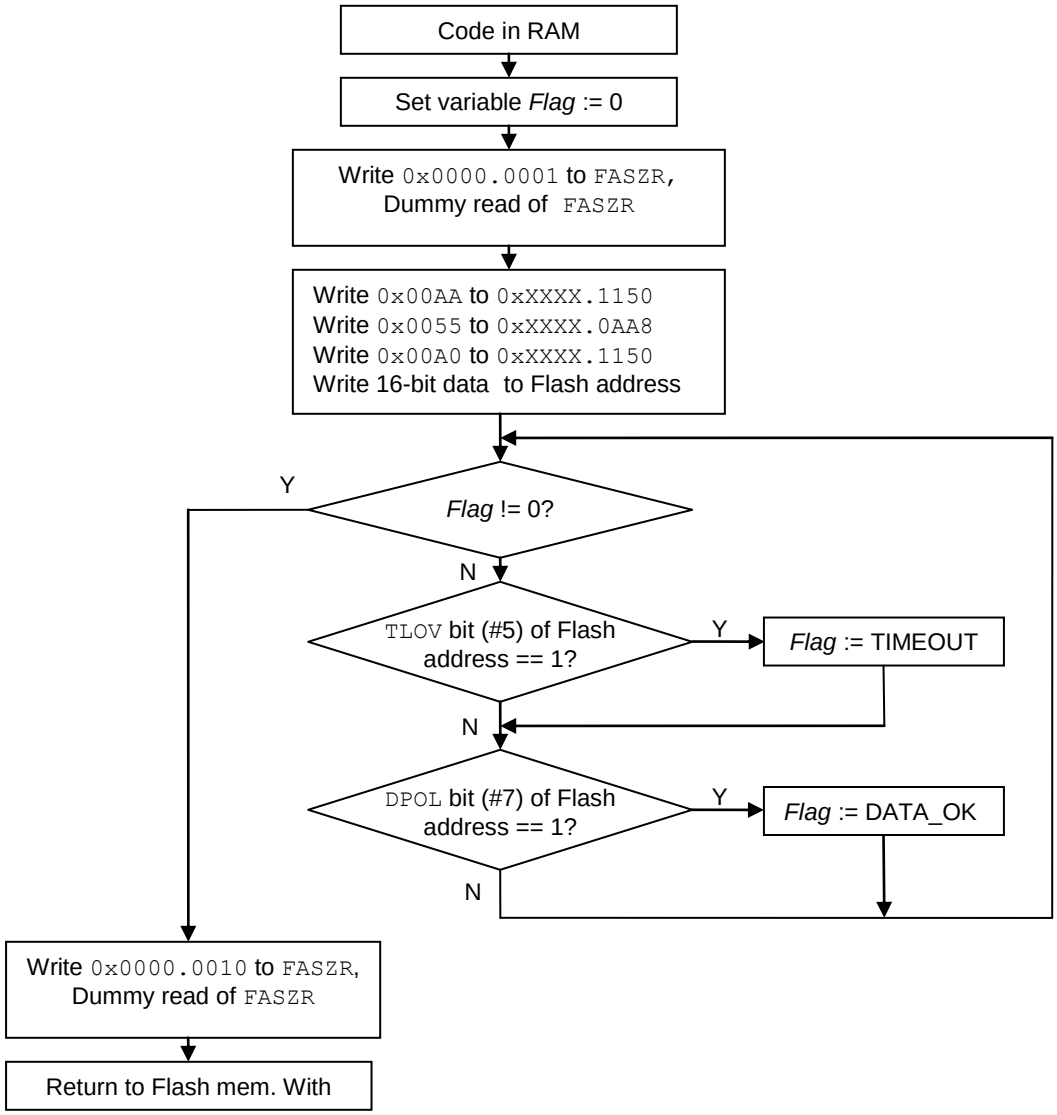
  FM3_FLASH_IF->FASZR &= 0xFFFE;      // ASZ[1:0] = 2'b10
  FM3_FLASH_IF->FASZR |= 0x2;
  u32DummyRead = FM3_FLASH_IF->FASZR; // dummy read of FASZR

  return (i32FlashFlag);
```

4.2 Main Flash Programming – Type 0 Devices

The following flowchart shows how to program a RAM code section for performing Flash 16-bit word programming.

Figure 8. Flash Sector Erase Type 0 Devices Flow Chart



Note: With Work Flash sequence addresses this flow diagram is also valid for Work Flash except that its execution does not need to be in RAM area.

The program code of the Flash programming RAM function may look like the following example:

Figure 9. Flash Sector Erase Type 0 Devices Software Example

```
#include "mb9bfxxx.h"

#define FLASH_SEQ_1550 ((volatile uint16_t*) 0x00001550) // sequence address 1
#define FLASH_SEQ_0AA8 ((volatile uint16_t*) 0x00000AA8) // sequence address 2

#define FLASH_WRITE_1 0x00AA   // flash write commands
#define FLASH_WRITE_2 0x0055
#define FLASH_WRITE_3 0x00A0

#define FLASH_DQ7 0x0080       // data polling flag bit (DPOL) position
#define FLASH_DQ5 0x0020       // time limit exceeding flag bit (TLOV) position

#define FLASH_TIMEOUT_ERROR -1
#define FLASH_OK             1

#ifdef __ICCARM__
  #pragma section = ".flash_ram_code"
#endif

#ifdef __ICCARM__
  __ramfunc
#elif  __CC_ARM
  __attribute__ ((section (".ramfunc")))
#else
  #error Please check compiler and linker settings for RAM code
#endif
int32_t FlashRomProgram(uint32_t u32ProgramAddress, uint16_t u16ProgamData)
{
  Volatile int32_t  i32FlashFlag = 0;
  Volatile uint32_t u32DummyRead;

  FM3_FLASH_IF->FASZR &= 0xFFFD;      // ASZ[1:0] = 2'b01
  FM3_FLASH_IF->FASZR |= 1;
  u32DummyRead = FM3_FLASH_IF->FASZR; // dummy read of FASZR

  *(FLASH_SEQ_1550) = FLASH_WRITE_1;
  *(FLASH_SEQ_0AA8) = FLASH_WRITE_2;
  *(FLASH_SEQ_1550) = FLASH_WRITE_3;
  *(volatile uint16_t*) u32ProgramAddress = u16ProgamData;

  while (0 == i32FlashFlag)
  {
    // Flash timeout?
    if((*(volatile uint16_t *)u32ProgramAddress & FLASH_DQ5) == FLASH_DQ5)
    {
      i32FlashFlag = FLASH_TIMEOUT_ERROR;
    }

    // Data correct?
    If ((*(volatile uint16_t *) u32ProgramAddress & FLASH_DQ7) == FLASH_DQ7)
    {
      i32FlashFlag = FLASH_OK;
    }
  }

  FM3_FLASH_IF->FASZR &= 0xFFFE;      // ASZ[1:0] = 2'b10
  FM3_FLASH_IF->FASZR |= 0x2;
  u32DummyRead = FM3_FLASH_IF->FASZR; // dummy read of FASZR

  return (i32FlashFlag);
}
```

Note: That the #pragma section directive is only needed once in a module for the IAR compiler.

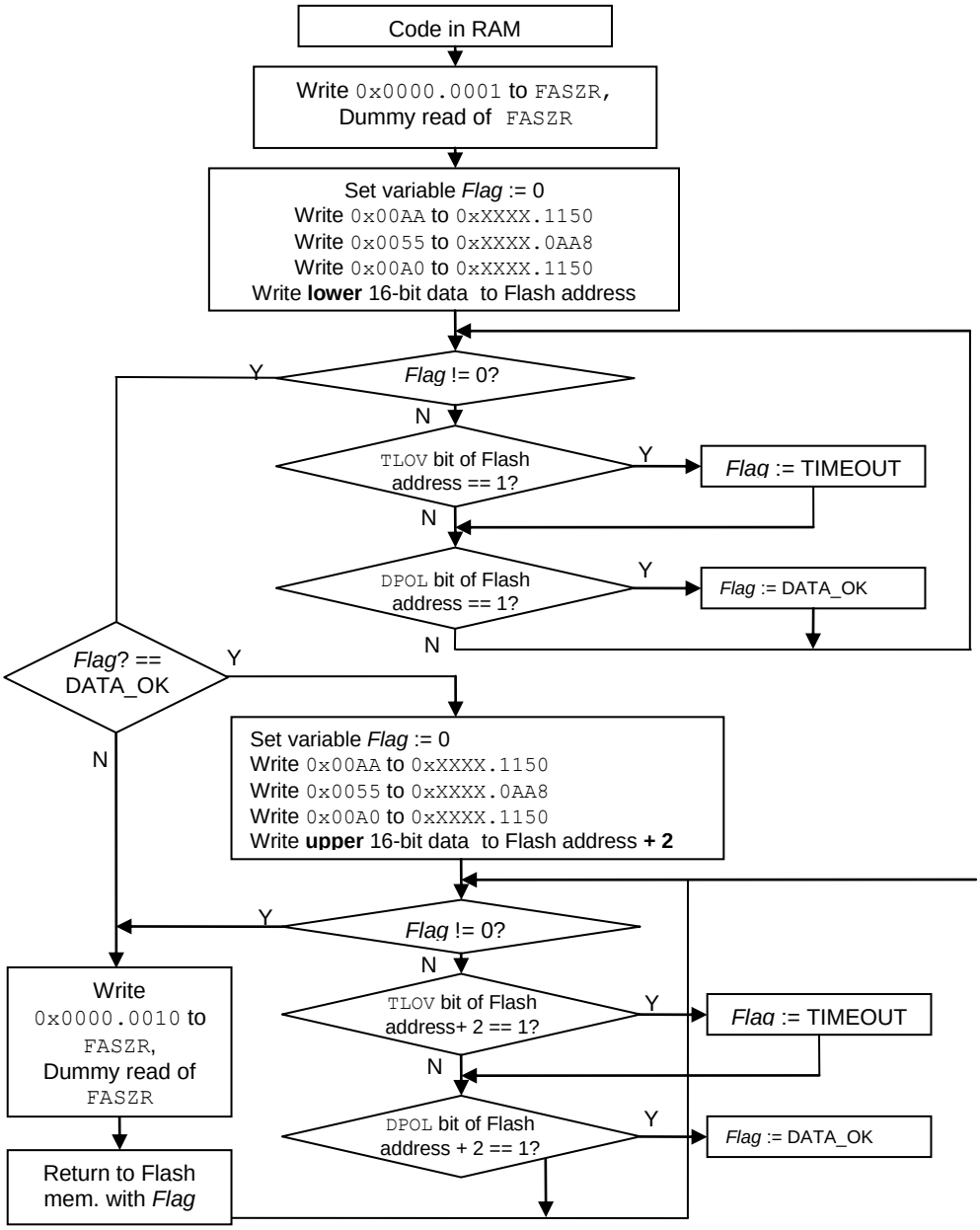
4.3 Type 1 Devices

For Type 1 devices the same algorithms can be used. The user only has to modify the sequence addresses according the tables shown in Table 1.

4.4 Main Flash Programming – Type 2 Devices

The following flowchart shows how to program a RAM code section for performing Flash 32-bit word programming

Figure 10. Flash Programming Type 2 Device Flow Chart



The program code of the Flash programming RAM function may look like the following example:

```
#include "mb9bfxxx.h"

#define FLASH_SEQ_1550 ((volatile uint16_t*) 0x00001550) // sequence address 1
#define FLASH_SEQ_0AA8 ((volatile uint16_t*) 0x00000AA8) // sequence address 2

#define FLASH_WRITE_1 0x00AA   // flash write commands
#define FLASH_WRITE_2 0x0055
#define FLASH_WRITE_3 0x00A0

#define FLASH_DQ7 0x0080       // data polling flag bit (DPOL) position
#define FLASH_DQ5 0x0020       // time limit exceeding flag bit (TLOV) position

#define FLASH_TIMEOUT_ERROR -1
#define FLASH_OK            1

#ifdef __ICCARM__
  #pragma section = ".flash_ram_code"
#endif

#ifdef __ICCARM__
  __ramfunc
#elif  __CC_ARM
  __attribute__ ((section (".ramfunc")))
#else
  #error Please check compiler and linker settings for RAM code
#endif
int32_t FlashDataPolling (uint32_t u32PollAddress, uint16_t u16PollData)
{
  volatile int32_t  i32FlashFlag = 0;
  volatile uint16_t u16DummyRead;

  u16DummyRead = *(volatile uint16_t *)u32PollAddress;
  while(0 == i32FlashFlag)
  {
    // Flash timeout?
    if((*(volatile uint16_t *)u32PollAddress & FLASH_DQ5) == FLASH_DQ5)
    {
      i32FlashFlag = FLASH_TIMEOUT_ERROR;
    }

    // Data correct?
    if((*(volatile uint16_t *)u32PollAddress & FLASH_DQ7) == (u16PollData & FLASH_DQ7))
    {
      i32FlashFlag = FLASH_OK;
    }
  }

  return i32FlashFlag;
}
```

Figure 11. Flash Programming Type 2 Devices Software Example

```
    // Data [0:15]
    u16HalfData = (uint16_t) (u32ProgamData & 0x0000FFFF);
    *(FLASH_SEQ_1550) = FLASH_WRITE_1;
    *(FLASH_SEQ_0AA8) = FLASH_WRITE_2;
    *(FLASH_SEQ_1550) = FLASH_WRITE_3;
    *(volatile uint16_t*)u32ProgramAddress = u16HalfData;
    i32FlashFlag = FlashDataPolling(u32ProgramAddress, u16HalfData);

    if (FLASH_OK == i32FlashFlag)
    {
      // Data [16:31] (Set ECC Flash cells)
      u16HalfData = (uint16_t) ((u32ProgamData >> 16) & 0x0000FFFF);
      *(FLASH_SEQ_1550) = FLASH_WRITE_1;
      *(FLASH_SEQ_0AA8) = FLASH_WRITE_2;
      *(FLASH_SEQ_1550) = FLASH_WRITE_3;
      *(volatile uint16_t*)(u32ProgramAddress + 2) = u16HalfData;
      i32FlashFlag = FlashDataPolling((u32ProgramAddress + 2),
  u16HalfData);
    }

    FM3_FLASH_IF->FASZR &= 0xFFFE;       // ASZ[1:0] = 2'b10
    FM3_FLASH_IF->FASZR |= 0x2;
    u32DummyRead = FM3_FLASH_IF->FASZR; // dummy read of FASZR

    return (i32FlashFlag);
  }
```

Note: That because the same data polling algorithm has to be used for the 1st lower 16-Bit word and the 2nd upper 16-Bit word, it was rolled out to an own C function (Flash Data Polling).

4.5 Project Adjustments for generating RAM Code and automatically Copying at Start-Up Phase

The following paragraphs will explain how to set up RAM code (for Main Flash only devices), which is copied from ROM to RAM at start-up phase, for the IAR and KEIL compilers.

4.5.1 IAR project settings

For compiling a RAM code section the user has to state a `#pragma section` directive with a section name, like *`.flash_ram_code`*. If different compilers should be able to compile the code, this `#pragama` directive has to be set in a `#ifdef __ICCARM__/#endif` pre-processor condition, which uses the predefined macro `__ICCRAM__` of the IAR compiler.

```
#ifdef __ICCARM__
  #pragma section = ".flash_ram_code"
#endif
```

Additionally the RAM functions itself shall get a special qualifier called `__ramfunc`. Also here place it in a pre-processor condition to avoid conflicts with other compilers.

```
#ifdef __ICCARM__
  __ramfunc
#endif
(Function declaration)
```

The last step is to adjust the linker file (<name>.icf). The following example shows in bold the additional lines for the RAM code linkage and automatically copying at start-up:

Figure 12. IAR Linker File

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml"
*/
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x00000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0007FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x1FFF8000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20007FFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0x800;
/**** End of ICF editor section. ###ICF###*/

define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__
to
    __ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__
to
    __ICFEDIT_region_RAM_end__];

define symbol __RAM_func_start__   = 0x20000000;
define symbol __RAM_func_end__     = 0x20007FFF;
define region RAM_func_region =   mem:[from __RAM_func_start__ to
__RAM_func_end__];

define block CSTACK    with alignment = 8, size =
__ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__
{ };

initialize by copy { readwrite };
do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section
.intvec };
define block RamCode { section .flash_ram_code };
place in RAM_func_region  { block RamCode };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
```

Note: That the blue highlighted name for the RAM code section must be the same stated in the #pragma section directive name attribute above.

With these settings the RAM code is placed in ROM, but copied automatically to RAM at start-up phase.

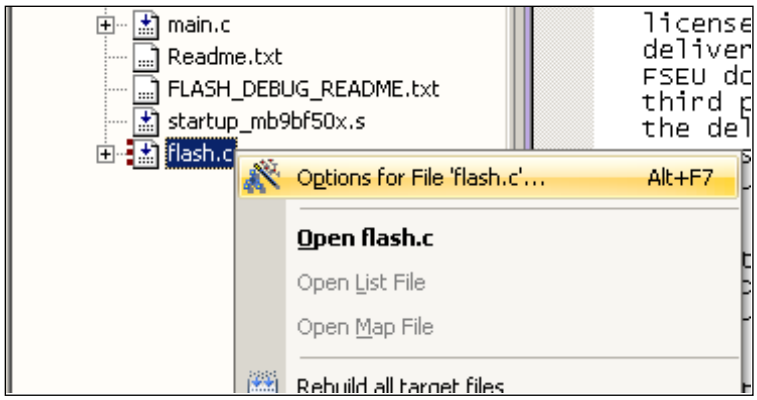
4.5.2 KEIL project settings

For compiling a RAM code function it should get the attribute __attribute__ ((section (".ramfunc"))) before its declaration. The KEIL compiler of the uVision IDE uses the predefined macro __CC_ARM for identification. The attribute shall be put within #ifdef __CC_ARM/#endif pre-processor condition to stay compatible with other compilers.

```
#ifdef __CC_ARM
  __attribute__ ((section (".ramfunc")))
#endif
```

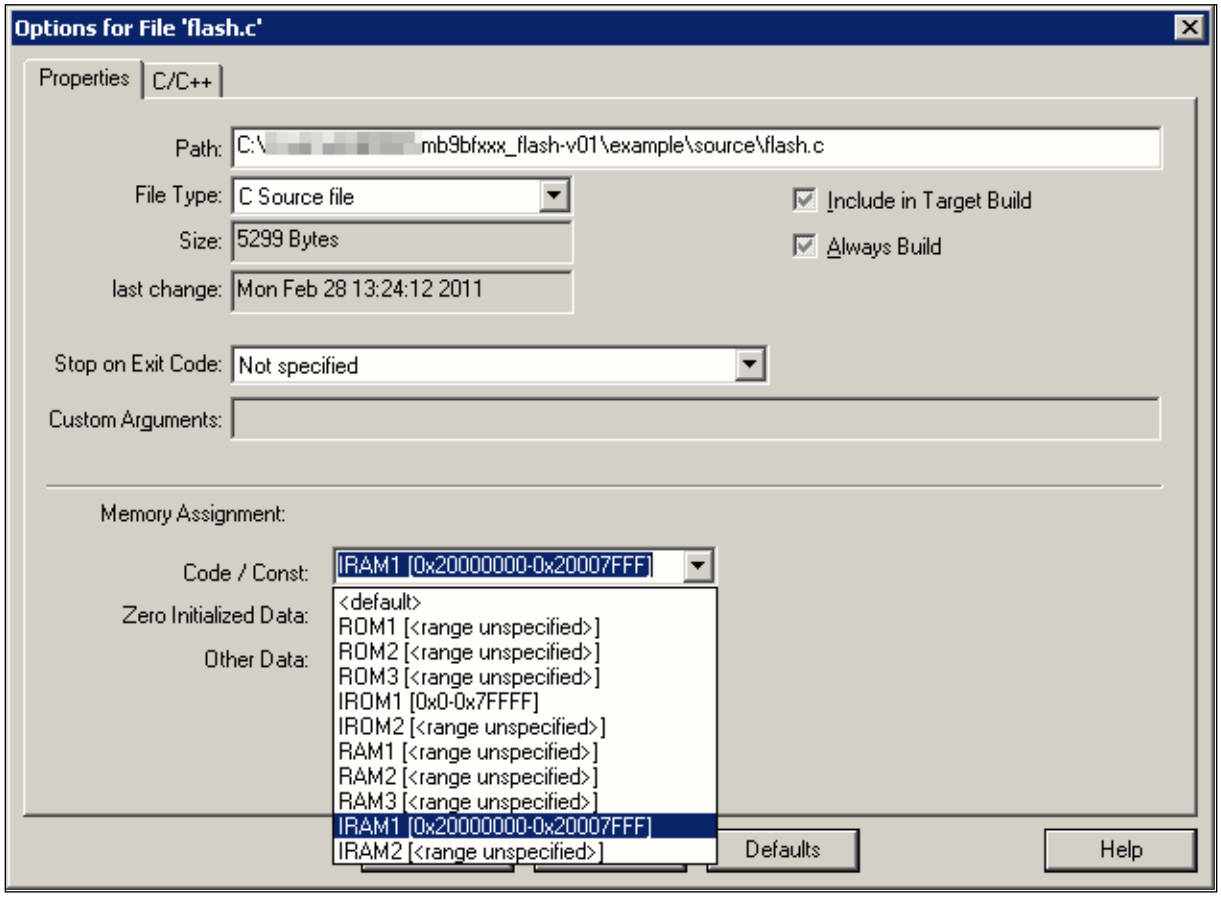
The linker settings can be done via the workspace tree. Please move the mouse cursor to the module which contain the RAM code functions, click on the right mouse button and choose “Options for File”.

Figure 13. KEIL uVision File Options Drop down Menu



In the options dialog, which follows, choose the tab “Properties” and adjust the Memory Assignment for Code / Const to an IRAM section.

Figure 14. uVision File Properties Dialog



Finally click on “OK”.

With this adjustment, the RAM code is compiled for RAM but linked to ROM automatically. It is copied at start-up phase from this ROM section to the IRAM section.

4.6 Intercompatibility for different Compilers

For writing RAM code which should be able to be compiled by different compiler the following example can be used. Assume that compiler 1 identifies itself by __ICC_ARM__ macro, compiler 2 by __CC_ARM, compiler 3 by __YAC_ARM__, and compiler 4 by __XYZ_ARM__.

Figure 15. Intercompatible Code Example

```
#ifdef __ICCARM__
  // individual setting, qualifier, directive, etc. for compiler 1
#elif  __CC_ARM
  // individual setting, qualifier, directive, etc. for compiler 2
#elif  __YAC_ARM__
  // individual setting, qualifier, directive, etc. for compiler 3
#elif  __XYZ_ARM__
  // individual setting, qualifier, directive, etc. for compiler 4
#else
  #error please check compiler and linker settings for RAM code
#endif
```

The #error directive is executed, if a compiler is used, which is not identified by the recent compilation process. This shows the user, that he has to take care for compiler-individual RAM code settings.

The linker settings must be done individually in any case by using different linker tools.

5 Document History

Document Title: AN204878 - FM3 MB9A/BFXXX with Flash Programming

Document Number: 002-04878

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	MAWI	02/28/2011	Initial release
			12/13/2011	Device type differences added
			03/16/2012	Work Flash added
*A	5053198	MAWI	12/16/2015	Migrated Spansion Application Note MCU-AN-300401-E-V12 to to Cypress format
*B	5875000	AESATMP9	09/07/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2011-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.