AN209372

F2MC-16FX Family 16-Bit Microcontroller All Series Start-up File

Related Application Notes: see Related Application Notes

All Microcontroller need special code to be executed before all others, which initializes all data areas generated by the compiler and which performs an application specific basic configuration of the microcontroller. This code is called Start-Up Code and is part of the Start-Up File.

Contents

1 Introduction....2
1.1 Main Features of the Start-Up File `start.asm`..2
1.2 Signpost....2
1.3 Overview of Start-Up File `start.asm`....2
2 User Settings....3
2.1 Controller Series, Device....3
2.2 C-language Memory Model....3
2.3 Constant Data Handling....3
2.4 Stack Type and Stack Size....4
2.5 General Register Bank....6
2.6 Low-Level Library Interface....6
2.7 Clock Selection....6
2.8 Clock Stabilization Time....7
2.9 External Bus Interface....7
2.10 ROM Mirror configuration....9
2.11 Flash Security....10
2.12 Flash Write Protection....10
2.13 Boot Vector....11
2.14 UART scanning....11
2.15 Enable RAMCODE Copying....11
2.16 Enable information stamp in ROM....11
2.17 Enable Background Debugging Mode....12
3 Configuration and Start-Up....12
3.1 Several fixed addresses....12
3.2 Declaration of __near addressed data sections 12
3.3 Declaration of RAMCODE section and labels...12
3.4 Declaration of sections containing other sections' descriptions....12
3.5 Stack area and stack top definition/declaration.13
3.6 Direct Page register dummy label definition.....13
3.7 Set Flash Security....13
3.8 Set Flash write protection....13
3.9 Debug address specification....14
3.10 Start-Up Code....14
4 Memory Models and Addressing Schemes....16
4.1 Far-Addressing....16
4.2 Near-Addressing....17
4.3 Direct addressing....18
4.4 Default Memory Models....18
5 Default Memory Sections....19
5.1 Data and Code Sections....20
5.2 Special Sections DCLEAR and DTRANS....21
5.3 Sections CINIT vs. CONST and RAMCONST vs. ROMCONST....21
6 Related Application Notes....22
6.1 Manuals and Application Notes....22
Worldwide Sales and Design Support....24
Cypress Developer Community....24
Technical Support....24

1 Introduction

All C-language applications need special code to be executed before all others, which initializes all data areas generated by the compiler and which performs an application specific basic configuration of the microcontroller. This code is called Start-Up Code and is part of the Start-Up File.

This document describes the Start-Up File `start.asm` for all F2MC16FX-microcontrollers.

The Start-Up File is included in the template project, which can be found on our webpage.

Please refer to Application Note AN204781 - F2MC-16FX Family, Softune Workbench for detailed information about where to get and how to use the template project.

1.1 Main Features of the Start-Up File `start.asm`

- Configuration part for easy step by step set-up of the Start-Up Code
- Configuration pre-configured and suitable for most types of single-chip applications
- Configuration allows to set Reset Vector with automatic mode byte setting
- Start-Up Code for pre-setting of core and external bus registers
- Start-Up Code for memory initialization according to F2MC-16 Family Softune™ C Compiler sections
- Start-Up File is linkage order independent
- Start-Up File is suitable for all memory models

1.2 Signpost

Readers who seek detailed description on how to configure and use the Start-Up File `start.asm` please refer to chapter 2 User Settings.

Readers who seek description on how the Start-Up File `start.asm` work in order to create their own Start-Up File please refer to chapter 3 Configuration and Start-Up and chapter 5 Default Memory Sections.

Though a bit out of scope, chapter 4 Memory Models and Addressing Schemes deals with memory addressing, which is fundamental to understand in all cases.

1.3 Overview of Start-Up File `start.asm`

The file start.asm consists of 6 major sections:

1. Contents
2. Disclaimer
3. History
4. Settings
5. Section and Data Declaration
6. Start-Up Code

Contents section gives a detailed overview over the file's contents in prose comments, Disclaimer states some legal issues and History contains the file's version history.

Settings is described in chapter 2 User Settings. Section and Data Declaration and Start-Up Code sections are described in chapter 3 Configuration and Start-Up.

2 User Settings

This chapter describes how to configure the Start-Up File.

The section 4 Settings in file start.asm is the only section that has to be edited by the user in order to set up a fully functional Start-Up File. All lines that are intended to be set by the user have a trailing comment that contains the string "<<<". When configuring a project, it is good practice to search for all occurrences of this string by the search function of your editor in order to check if all necessary settings are correct.

This chapter describes all paragraphs of the User Settings' section.

2.1 Controller Series, Device

This paragraph lets you choose the series and device that is to be used for the project. You can choose from the series and devices listed.

If you are using an emulator system (V-chip), then choose the device you want to emulate.

2.2 C-language Memory Model

The memory model describes how data and code are accessed by compiler-generated assembler code. Please refer to the chapter 4 Memory Models and Addressing Schemes in order to understand the mechanism. Available options are SMALL, MEDIUM, COMPACT and LARGE.

Within the Start-Up File this setting only affects the type of CALL-instruction that is used to call library functions or application *main()* function from Start-Up File. This is either a physical CALLP (24-bit address) for LARGE and MEDIUM setting or a CALL (16-bit offset only) for SMALL and COMPACT setting.

Note that the Start-Up File setting does not affect the actual memory model that has to be used by the compiler. Therefore, the correct compiler setting still is mandatory.

The setting AUTOMODEL generates always 24-bit address CALLP-instructions in the Start-Up File. If the main module (containing *main()* function) is compiled with Small or Compact model or if libraries for those models are used, they will return with RET-instruction (removes 2 address Bytes from stack) and not with RETP-instruction (removes 4 address Bytes from stack). The remaining two Bytes on stack are automatically removed by the Start-Up Code. Therefore, AUTOMODEL is working with all memory models.

The Start-Up File setting MEMMODEL does not affect data accesses and data initializations. Regardless of the memory model all section types are always initialized (except *CINIT*, see 5 Default Memory Sections).

If *AUTOMODEL* setting is used, the Start-Up File has not to be changed or modified, if the compiler memory model is changed. AUTOMODEL is the recommended setting.

2.3 Constant Data Handling

This paragraph specifies how 16-bit addressed constant data are handled by the Start-Up File. Please refer to the chapters 4 and 5 also in order to understand the mechanism.

CONSTDATA controls whether the constants are actually copied from CONST section (ROM) to CINIT section (RAM) by Start-Up Code. The setting does not affect the compiler option RAMCONST. It is still mandatory to set the compiler to the correct mode.

In general, the option RAMCONST means that 16-bit addressed data are copied from ROM to RAM in order to be addressed by 16 Bit offset only. In this case the Start-Up File will declare CINIT section and initialize it.

ROMCONST means that these data are not copied but accessed in ROM by 16-bit offset only. In this case the Start-Up File will not declare the CINIT section and it will not execute the copy routine. This mode is intended for use together with the ROM Mirror functionality, described in 2.10 ROM Mirror configuration.

Recommended Start-Up File option is RAMCONST because this works regardless of the compiler setting. If the compiler is set to ROMCONST and the Start-Up File is set to RAMCONST, the code for copying the constant data is executed for a section of size 0, resulting in no data copied.

2.4 Stack Type and Stack Size

This paragraph specifies the configuration of the stacks, when the application *main()*-function is called.

2.4.1 Stack Type

F²MC16-microcontrollers provide two different types of stack: System Stack and User Stack. The application can use either System Stack only or it can use both System Stack and User Stack. The type of stack is specified by the setting *STACKUSE*. It can either be *USRSTACK* or *SYSSTACK*.

System Stack (option *SYSSTACK*) is always used for interrupt function. It is automatically selected, if hardware interrupts are executed or if software interrupts are called. All stack operation of interrupt handlers will work with the System Stack.

Outside of interrupt handlers the pre-selected stack type is used. This is either User Stack (option *USRSTACK*) or System Stack.

If *SYSSTACK* is set, only the System Stack area has to be prepared. All operation will work on the same stack. The necessary safety margin has to be reserved for one stack only. *SYSSTACK* should be used, if the necessary RAM consumption for stack has to be low.

Note: Even if *SYSSTACK* is selected, a User Stack section will be allocated. The User Stack may have a size of 0 Byte. The User Stack Bank Register (*USB*) will be initialized to the Bank where the User Stack is allocated.

If *USRSTACK* is set, both System Stack and User Stack have to be prepared. The necessary safety margin has to be provided twice. *USRSTACK* should be used, if separated stack areas are necessary for management reasons. This might be the case with schedulers, operating systems or other applications.

Note: The *F2MC-16 Family Softune™ C Compiler* assumes the setting *USRSTACK* when performing pointer type casts from local variables to `__far` qualified pointers. I.e. Stack relative Addresses are expanded by

System Stack Bank (*SSB*) if the containing function is `__interrupt` qualified,

System Stack Bank (*SSB*) if the containing function was compiled with `#pragma SSB`,

Either System Stack Bank (*SSB*) or User Stack Bank (*USB*) depending on the current active stack (*S-Flag* of CPU) if the containing function was compiled with `#pragma except` (Involves a call to system library),

User Stack Bank (*USR*) in all other cases (default).

Therefore at least one of the following restrictions has to be met:

User Stack and System Stack are located on the same bank. (recommended)

SYSSTACK is selected and all functions are compiled with `#pragma SSB`

All functions that are called only from interrupt contexts are compiled with `#pragma SSB` and all functions that may be called from arbitrary context are compiled with `#pragma except`.

2.4.2 Stack Size

The *STACK_RESERVE* option controls if the Start-Up File will declare memory sections for the stacks (option *ON*), or if this is done in another file (option *OFF*).

Recommended setting for *STACK_RESERVE* is *ON*. In this case the size of both stacks can be controlled by *STACK_SYS_SIZE* (size of System Stack) and *STACK_USR_SIZE* (size of User Stack).

These settings specify the amount of Bytes to be reserved for stacks. The stacks have to cover all:

- parameters passed over stack
- return addresses for function calls
- local variables and constants (defined in function context)
- temporary data due to compiler optimization
- interrupt context on System Stack
- safety margin

For estimating necessary stack size the compiler offers the `-INF stack` option. With it the compiler generates stack information files (extension `stk`), which list the number of Bytes necessary to execute each function. These stack information files are already available for all library functions and can be found in the `Lib\907\` subdirectory of the Softune installation.

2.4.3 Stack Pattern

If the *STACK_RESERVE* option is enabled the reserved stacks can be filled with a pattern at start-up. This can be a help when debugging stack related problems.

The stacks will be filled with a word-pattern (16 bit) defined by *STACK_PATTERN*, when both options *STACK_RESERVE* and *STACK_FILL* are enabled.

2.5 General Register Bank

The setting *REGBANK* specifies the Register Bank that is configured, when the application *main()*-function is called.

F2MC16-microcontrollers provide 32 general purpose Register Banks of 16 Bytes each. The current Register Bank is selected by the *Register Pointer RP* (5 bit), which is part of the dedicated core register *Processor State PS*.

Usually the default Bank is Bank 0.

Note: This Start-Up File does not reserve the RAM area for the Register Bank. Reserving the area for all used Register Banks is a mandatory setting of the linker.

2.6 Low-Level Library Interface

The setting *CLIBINIT* specifies whether the Start-Up Code has to call the stream initialization function of the C-library.

The stream initialization is necessary only, if streamed IO-functions are used. These functions (e.g. `printf()`) also require the definition of application specific low-level functions. For other library functions (like e.g. `sprintf()`) all this is not necessary. However, several functions consume a large amount of stack. For more information refer to the compiler help.

2.7 Clock Selection

This paragraph specifies the clock settings that are applied before calling application *main()*- function.

The setting *CRYSTAL* specifies the frequency of the external oscillator. The Start-Up File (version 1.31) does support two frequency settings for the external crystal: 4 MHz (setting *FREQ_4MHZ*) or 8 MHz (setting *FREQ_8MHZ*).

Setting *CLOCK_SPEED* specifies the peripheral clocks *CLKP1* and *CLKP2*. *CLKP2* is used for *CAN Controllers* and *Sound Generators* only. All other peripherals use *CLKP1*. Possible settings for *CLOCK_SPEED* are listed in Table 1.

Table 1. Clock frequency settings

Setting	CPU frequency	CLKP1 frequency	CLKP2 frequency
CPU_4MHZ_MAIN_CLKP2_4MHZ	4 MHz Main Clock	4 MHz Main Clock	4 MHz Main Clock
CPU_4MHZ_PLL_CLKP2_4MHZ	4 MHz PLL	4 MHz PLL	4 MHz PLL
CPU_8MHZ_CLKP2_8MHZ	8 MHz	8 MHz	8 MHz
CPU_12MHZ_CLKP2_12MHZ	12 MHz	12 MHz	12 MHz
CPU_16MHZ_CLKP2_16MHZ	16 MHz	16 MHz	16 MHz
CPU_24MHZ_CLKP2_12MHZ	24 MHz	24 MHz	12 MHz
CPU_32MHZ_CLKP2_16MHZ	32 MHz	32 MHz	16 MHz
CPU_32MHZ_CLKP1_16MHZ_CLKP2_16MHZ	32 MHz	16 MHz	16 MHz
CPU_48MHZ_CLKP2_16MHZ	48 MHz	48 MHz	16 MHz
CPU_48MHZ_CLKP1_32MHZ_CLKP2_16MHZ	48 MHz	32 MHz	16 MHz
CPU_56MHZ_CLKP2_14MHZ	56 MHz	56 MHz	14 MHz

The clock is set quite early. However, if *CLOCKWAIT* is *ON*, polling for machine clock to be switched to *Main Clock* or *PLL* is done at the end of Start-Up File. Therefore, the stabilization time is not wasted. *main()*-function will finally start at correct speed. Resources can be used immediately.

2.8 Clock Stabilization Time

Ceramic and crystal oscillators generally require several ms to stabilize at their natural frequency (oscillation frequency) when oscillation starts. The internal *PLL* multiplier circuit and the *RC Oscillator* also need a certain stabilization time after activation. For this reason, *CPU* operation with the activated clock is not allowed immediately after oscillation starts but is allowed only after full oscillation stabilization. Because the oscillation stabilization time of the *Main* and *Sub oscillator* depends on the type of oscillator (crystal, ceramic, etc.), the proper oscillation stabilization wait interval for the used oscillator must be selected.

RC Clock stabilization time is fixed by hardware to 764 respectively 64 *RC Clock* cycles depending on the reset cause. After hardware reset *RC Clock* stabilization time is approximately 382 µs. The CPU will not operate at all before this time has elapsed.

PLL Clock stabilization time is fixed to 2^{14} Main Clock cycles by the Start-Up File, resulting in approximately 4 ms stabilization time at 4 MHz Main clock frequency (2 ms at 8 MHz Main Clock).

Sub Clock stabilization time is fixed to 2^{16} Sub Clock cycles by the Start-Up File, resulting in approximately 2 s stabilization time at 32 kHz Sub Clock frequency. Note that this setting is ignored for devices that do not support a Sub Clock.

Main Clock stabilization time can be selected by *MC_STAB_TIME*. Possible values are listed in Table 2.

Table 2. Main Clock stabilization time settings

Setting	Cycles of Main Clock	approximate Time at 4 MHz Main Clock	approximate Time at 8 MHz Main Clock
MC_2_10_CYCLES	2^{10}	256 µs	128 µs
MC_2_12_CYCLES	2^{12}	1 ms	512 µs
MC_2_13_CYCLES	2^{13}	2 ms	1 ms
MC_2_14_CYCLES	2^{14}	4 ms	2 ms
MC_2_15_CYCLES	2^{15}	8 ms	4 ms
MC_2_16_CYCLES	2^{16}	16 ms	8 ms
MC_2_17_CYCLES	2^{17}	32 ms	16 ms
MC_2_18_CYCLES	2^{18}	65 ms	32 ms

2.9 External Bus Interface

The *External Bus Interface* can be used to connect external devices via the parallel bus interface. By doing so, the address space of external devices can be bond to the microcontroller's memory map. This behavior must be initialized at start-up.

Details on the *External Bus Interface* can be found in Application Note *300208 "16FX External Bus Interface"*.

2.9.1 BUSMODE

This setting specifies the use of external bus interface. For all devices without external bus interface it has to be set to *SINGLE_CHIP*.

If *BUSMODE* is set to *SINGLE_CHIP*, all further settings of the External Bus Interface section are ignored.

If *BUSMODE* is set to *INTROM_EXTBUS*, the internal ROM (Mask, Flash) is still accessible. If *EXTROM_EXTBUS* is selected the internal ROM is not accessible. Detailed address specification can be found in the controller Hardware Manual.

If *BUSMODE* is set to *INTROM_EXTBUS* or *EXTROM_EXTBUS* the external bus interface is enabled. In this case, it has to be configured properly. Please refer to the Microcontroller Hardware Manual for detailed information.

2.9.2 ADDRESSMODE

Some devices support multiplexed and/or non-multiplexed Bus Mode. Please refer to the related datasheet / Hardware Manual.

For those devices that support non-multiplexed Bus Mode, the setting *NON_MULTIPLEXED* can be selected in order to enable non-multiplexed Mode.

The multiplexed Mode is available for all devices that support the External Bus Interface. It can be enabled by selecting *MULTIPLEXED*.

2.9.3 External Bus Areas

The 16MB address area of the F2MC-16FX MCU can be divided into 6 External bus areas with dedicated configuration of *Chip Select* signals. The address ranges of the external bus areas 0 and 1 are fixed, while the address ranges of the other areas can be programmed within the upper 15MB address space as shown on Table 3.

The settings *CS0_CONFIG* to *CS5_CONFIG* specify the *Automatic Wait Cycles*, *Address Cycle Extension*, *Strobe timing*, *Write Strobe Function*, *Endianness*, *Bus Width*, *Chip Select Output Enable* and *Chip Select Level*. For details please refer to the Hardware Manual.

Table 3. External Area

External Area	Available Address Area	Start Bank of Chip Select Area
0	0x0000F0 to 0x0000FF (fixed)	Fixed
1	0x000C00 to Start of RAM – 1 (see datasheet for exact address)	Fixed
2	0x100000 to 0xFFFFFF	CS2_START
3		CS3_START
4		CS4_START
5		CS5_START

Each of the 6 memory areas can be activated by the options *CHIP_SELECT0* to *CHIP_SELECT5*. Memory access outside activated address areas does not show any activity on the I/O cells associated to the external bus interface.

2.9.4 Signal Selection

Some of the bus signals are optional and can either be enabled or disabled. The address lines are controlled by the settings *ADDR_PINS_23_16*, *ADDR_PINS_15_8* and *ADDR_PINS_7_0*, each bit controlling a dedicated address line. The other Signals are listed on Table 4.

Table 4. Optional Signals for External Bus

Option name	Description	Pin name
LOW_BYTE_SIGNAL	Low Byte Signal	LBX
HIGH_BYTE_SIGNAL	High Byte Signal	UBX
LOW_WRITE_STROBE	Write Strobe Signal	WRLX/WRX
HIGH_WRITE_STROBE	Write Strobe Signal	WRHX
READ_STROBE	Read Strobe Signal	RDX
ADDRESS_STROBE	Address Strobe Signal	ALE/ASX

2.9.5 Bus Options

Table 5 lists the general options for the External Bus Interface. Please refer to the Hardware Manual for details.

Table 5. Options for External Bus

Option name	Description
HOLD_REQ	Enable/Disable Hold Function
EXT_READY	Enable/Disable External Ready Function
EXT_CLOCK_ENABLE	Enable/Disable External Bus Clock Output
EXT_CLOCK_INVERT	Enable/Disable Clock Inversion
EXT_CLOCK_SUSPEND	Enable/Disable suspending of external clock when no transfer in progress.
ADDRESS_STROBE_LVL	Address Strobe Function: OFF - active low; ON - active high
EXT_CLOCK_DIVISION	CLKB frequency is divided by this value to get the frequency of the external bus clock

2.10 ROM Mirror configuration

This paragraph specifies the usage of the ROM Mirror function. This function mirrors an area in ROM to Bank 0, so that this area can be accessed by 16-bit near Pointers. This function is intended to be used for access of constant data, which then can be accessed the same way as non-constant variables in internal RAM (by near pointers and Data Bank Register set to 0) are accessed, without actually wasting RAM.

If *ROMIRROR* is set to *ON*, any access to an address in the mirrored area in Bank 0 is redirected to the mirrored Bank with same 16-bit offset. The mirrored area is fixed to the end of both Banks.

The ROM Bank size of the mirrored area can be selected by *MIRROR_SIZE*. The Target Bank is specified by *MIRROR_BANK + 0xF0*.

Table 6 shows possible configurations.

Table 6. ROM Mirror Areas

MIRROR_SIZE setting	Mirrored Area (for read access)	Mirror Area (ROM data) X=MIRROR_BANK
MIRROR_8KB	0x00E000..0x00FFFF	0xFXE000..0xFXFFFF
MIRROR_16KB	0x00C000..0x00FFFF	0xFXC000..0xFXFFFF
MIRROR_24KB	0x00A000..0x00FFFF	0xFXA000..0xFXFFFF
MIRROR_32KB	0x008000..0x00FFFF	0xFX8000..0xFXFFFF

Recommended setting is ON because it allows the usage of the *ROMCONST*-option of the compiler in internal-ROM modes (see 5.3 Sections CINIT vs. CONST and RAMCONST vs. ROMCONST).

2.11 Flash Security

This paragraph specifies the *Flash Security* settings. These settings are stored to dedicated addresses in Flash and prevent the Flash from being read by any other means than the internal application.

If the application is started in Internal Vector Mode, the Flash is accessible without restrictions. If the application is started by *External Boot Vector Fetch* (modes 0/1/6), *External Parallel Flash Programmer* (mode 7) or *Serial Communication Mode* (mode 2) the access to Flash is restricted to Chip Erase (erase all Flash contents). However, in *Serial Communication* mode the security protection can be disabled by a 128-bit security key, if configured.

Some devices have a *Flash B* (formerly Satellite Flash), which can be accessed independently to *Flash A* (formerly Main Flash). This *Flash B* has its own security settings. If the device has a *Flash B* the option *FLASH_B_AVAILABLE* should be enabled. If this option is disabled, all further settings regarding *Flash B* are ignored.

Options *FLASH_A_SECURITY_ENABLE* and *FLASH_B_SECURITY_ENABLE* control whether the security feature shall be enabled or disabled for *Flash A* respectively *Flash B*.

The unlock key can be specified by *FLASH_A_UNLOCK_0* to *FLASH_A_UNLOCK_15*, respectively *FLASH_B_UNLOCK_0* to *FLASH_B_UNLOCK_15*.

2.12 Flash Write Protection

Flash Memory can be protected from mistakenly being programmed or erased by setting up a *Write Protection*. The *Write Protection* can be specified for each sector separately.

Initialization of *Flash A* Write Protection Registers at start-up can be achieved by setting *FLASH_A_WRITE_PROTECT* to *ON*, initialization of *Flash B* Write Protection Registers at start-up can be achieved by setting *FLASH_B_WRITE_PROTECT* to *ON*. The settings for protection of individual sectors are listed on Table 7.

Table 7. Flash Sector Write Protection

Option	Address area of protected sector
PROTECT_SECTOR_SA0	0xDF0000..0xDF1FFF
PROTECT_SECTOR_SA1	0xDF2000..0xDF3FFF
PROTECT_SECTOR_SA2	0xDF4000..0xDF5FFF
PROTECT_SECTOR_SA3	0xDF6000..0xDF7FFF
PROTECT_SECTOR_SA32	0xF80000..0xF8FFFF
PROTECT_SECTOR_SA33	0xF90000..0xF9FFFF
PROTECT_SECTOR_SA34	0xFA0000..0xFAFFFF
PROTECT_SECTOR_SA35	0xFB0000..0xFBFFFF
PROTECT_SECTOR_SA36	0xFC0000..0xFCFFFF
PROTECT_SECTOR_SA37	0xFD0000..0xFDFFFF
PROTECT_SECTOR_SA38	0xFE0000..0xFEFFFF
PROTECT_SECTOR_SA39	0xFF0000..0xFFFFFF
PROTECT_SECTOR_SB0	0xDE0000..0xDE1FFF
PROTECT_SECTOR_SB1	0xDE2000..0xDE3FFF
PROTECT_SECTOR_SB2	0xDE4000..0xDE5FFF
PROTECT_SECTOR_SB3	0xDE6000..0xDE7FFF

2.13 Boot Vector

This setting specifies the generation of a Boot Vector. A Boot Vector contains the start address of the program.

There are three possible settings for *BOOT_VECTOR*: *OFF*, *BOOT_VECTOR_TABLE* and *BOOT_VECTOR_FIXED*.

If *BOOT_VECTOR* is set to *OFF*, Start-Up File does not create any Boot Vector related configurations, they have to be stated by the application instead.

If *BOOT_VECTOR* is set to *BOOT_VECTOR_TABLE*, the start address of the Start-Up File is written to the vector table by adding them to linker sections *RESVECT* and *BOOT_SELECT*. Note that section *RESVECT* will intersect with section *INTVECT*, which is created by the compiler from all `#pragma intvect` instructions, if interrupts 0..7 are specified. This is not the case for file `vectors.c` from template project, i.e. the file can be used together with setting *BOOT_VECTOR_TABLE*.

If *BOOT_VECTOR* is set to *BOOT_VECTOR_FIXED*, a marker is set in ROM Configuration Block (Bank 0xDF) and the code at address 0xDF0080 is started. This way a *Boot Loader* can be placed to the small sectors at Bank 0xDF, which will be started without any access to the vector table (Bank 0xFF).

2.14 UART scanning

The setting *UART_SCANNING* specifies if the MCU shall scan for UART communication in *Internal Vector Mode* during some milliseconds after start-up. If enabled, the device can be programmed without switching to *Serial Communication Mode* by simple reset. If disabled programming is possible in *Serial Communication Mode* only. However, UART scanning slows down start-up of the MCU.

2.15 Enable RAMCODE Copying

The setting *COPY_RAMCODE* specifies if code shall be copied from ROM to RAM during start-up.

If enabled, the code to be copied must be linked to section RAMCODE (e.g. by `#pragma section FAR_CODE=RAMCODE`). Note that the code must be defined far (e.g. by setting default memory model to medium or large). The copy-routine looks for two sections: *RAMCODE* (target location in RAM) and *@RAMCODE* (source location in ROM). Both sections have to be defined by linker settings.

Note: The linker will treat any code in section RAMCODE as if it was linked to section *RAMCODE* in RAM, but will store it to section *@RAMCODE* in ROM. All jump addresses within that code point to RAM.

Note: The code linked as RAMCODE must be copied to the exact location of *RAMCODE* in RAM. The code is not independent of the link location. Unless *RAMCODE* and *@RAMCODE* are linked to the same Bank Offset and *near*[1] addressing is used, the code is not executable at its location at *@RAMCODE* in ROM.

2.16 Enable information stamp in ROM

Option *VERSION_STAMP* specifies whether the version number of the Start-Up File shall be stored to ROM. If enabled, section *VERSIONS* is created that contains a new-line and zero terminated string specifying the version in a human readable manner.

[1] See chapter 4 Memory Models and Addressing Schemes.

2.17 Enable Background Debugging Mode

Most F2MC16FX-microcontrollers can be configured for background debugging. Therefore a debugging-tool is needed to control and monitor the MCU. Please refer to the appropriate Application Note for details on using and configuring Background Debugging Mode. Currently there is one such tool available:

- EUROScope from EUROS Embedded Systems GmbH, Application Note AN204824 - F2MC-16FX FAMILY, SETUP AND DEBUGGING WITH EUROSCOPE

If you don't want to use the Background Debugging feature, set *BACKGROUND_DEBUGGING* to *OFF*. All further settings regarding the *Background Debugging Mode* will be ignored then.

3 Configuration and Start-Up

This chapter describes what the Start-Up File makes of the provided configuration parameters.

The section *Section* and *Data Declaration* states all non-executable settings according to the configuration of section Settings. The section *Start-Up Code* states all commands executed during start-up according to the configuration of section *Settings*. These sections are not intended to be modified by the user. Hence it is not necessary to understand this chapter in order to configure the Start-Up File.

The reader targeted by this chapter wants to understand the Start-Up File in order to write a custom Start-Up File.

3.1 Several fixed addresses

This paragraph declares some register names that are used in the executed part of the Start-Up File and defines their addresses. The register names are analogous to those mentioned in the Hardware Manual.

3.2 Declaration of __near addressed data sections

This paragraph declares the memory sections *DATA*, *DATA2*, *DIRDATA*, *LIBDATA*, *INIT*, *INIT2*, *DIRINIT*, *LIBINIT*, *CINIT*, *CINIT2*, *DCONST*, *DIRCONST*, *LIBDCONST*, *CONST* and *CONST2*. The sections suffixed with "2" are treated equally to those without the suffix, but can be used for a separate application (e.g. Boot Loader). The purpose of the non-suffixed sections is explained in chapter 5 Default Memory Sections.

Note that sections *CINIT* and *CINIT2* are not declared if *CONSTDATA* is set to anything other than *RAMCONST*.

3.3 Declaration of RAMCODE section and labels

If *COPY_RAMCODE* is enabled this paragraph imports the compiler symbols *_RAM_RAMCODE* and *_ROM_RAMCODE*. The linker generates these symbols by linking section *RAMCODE* to both linker sections *RAMCODE* and *@RAMCODE*. *_RAM_RAMCODE* is the start address of the section when linked to section *RAMCODE* (RAM) and *_ROM_RAMCODE* is the start address of the section when linked to section *@RAMCODE* (ROM).

Note: Compiler symbols prefixed by *_RAM_* and *_ROM_* are generated for all pairs of sections with same names (defined by the "-sc" directive) except for the @-sign that prefixes one of the sections. The compiler symbol prefixed by *_RAM_* will be linked to the non-prefixed liker section, whereas the compiler symbol prefixed by *_ROM_* will be linked to the @¬-prefixed linker section.

3.4 Declaration of sections containing other sections' descriptions

This paragraph declares sections *DCLEAR*, which contains information about all sections that have to be cleared to 0 at start-up, and *DTRANS*, which contains information about all sections that have to be filled with initial data that is stored to ROM.

Please refer to *section 5.2 Special Sections DCLEAR and DTRANS* for details.

3.5 Stack area and stack top definition/declaration

This paragraph declares the stack sections *SSTACK* and *USTACK* and reserves memory for them, if configured by *STACK_RESERVE*.

If memory is reserved, the compiler symbols *__systemstack*, *__systemstack_top*, *__userstack* and *__userstack_top* are exported. If not, the symbols are imported and have to be defined by another module.

Symbols *__systemstack* and *__userstack* define the lowest address of the stacks. Symbols *__systemstack_top* and *__userstack_top* define the address after the end of the stacks. Since the stacks are filled from high address to low address, the stacks will be initialized relative to addresses *__systemstack_top* and *__userstack_top*.

3.6 Direct Page register dummy label definition

This paragraph defines section *DIRDATA* and label *DIRDATA_S*, which is appended to the section.

This label is used to get the Page of the direct data. Depending on the linkage order of this Start-Up File the label is placed anywhere within the direct data Page. However, the statement `PAGE (DIRDATA_S)` is processed. Therefore, the lower 8 bits of the address of *DIRDATA_S* are not relevant and this feature becomes linkage order independent.

Note: the linker settings have to make sure that all direct data are located within the same physical Page (256 Byte block).

3.7 Set Flash Security

This paragraph sets the ROM Configuration Block for Flash Security. The *ROM Configuration* Block is located at fixed addresses in flash. Its security related parts are located at `0xDF0000..0xDF0011` for *Flash A* and if available `0xDE0000..0xDE0011` for *Flash B*. Both address ranges are structured similar and each controls only the Flash Macro it is located in.

Note: *Flash B* is not secured by the security settings stated in *Flash A* and vice versa.

The security related part is structured in the following way:

- 1 Byte Magic Word 0x99
- 1 Byte reserved
- 16 Bytes Security Key

If the first Byte contains anything other than the Magic Word 0x99, Flash is unsecured and the following 17 Bytes are not evaluated. Hence they can be used as general purpose ROM.

Note: By erasing the sector that contains the security information, the whole Flash Macro (whole Flash A or whole Flash B) is unsecured, since the first Byte become 0xFF.

3.8 Set Flash write protection

This paragraph sets the *ROM Configuration Block* for *Flash Write Protection*. The *ROM Configuration Block* is located at fixed addresses in flash. Its write protection related parts are located at `0xDF001C..0xDF0024` for *Flash A* and if available `0xDE0000..0xDE0024` for *Flash B*. Both address ranges are structured similar and each controls only Flash Macro (Flash A or Flash B) it is located in.

Note: *Flash B* is not protected by the write protection settings stated in *Flash A* and vice versa.

The write protection related part is structured in the following way:

- 4 Bytes Magic Word 0x7B, 0x3A, 0x2D, 0x29
- 40 bits controlling write protection of the sectors SA0..SA39 respectively SB0..SB39. Any bits corresponding to not existing sectors are ignored. A value of 1 means access will be granted, 0 means access will be protected.

If the first four Bytes contain anything other than the Magic Word 0x7B, 0x3A, 0x2D, 0x29, Flash will be unprotected and the following 5 Bytes will not be evaluated. Hence they can be used as general purpose ROM.

Note: By erasing the sector that contains the security information, the whole Flash Macro (whole Flash A or whole Flash B) is unprotected, since the first four Bytes become 0xFF.

3.9 Debug address specification

This paragraph sets the *ROM Configuration Block* for *Background Debugging*. The *ROM Configuration Block* is located at fixed addresses in flash. Its background debugging related parts are located at `0xDF0040..0xDF0024`.

The Background Debugging related part is structured in the following way:

- 4 Bytes Magic Word 0x7B, 0x3A, 0x2D, 0x29
- 2 Bytes BDM Configuration
- 2 Bytes Baud Rate Divider
- 3 Bytes External Address Marker
- 1 Byte Watchdog Pattern
- 4 Bytes reserved
- 8 Bytes Initial Patch Function Control Status
- 12 Bytes Initial Patch Function Address
- 8 Bytes Initial Patch Function Data

For details please refer to Application Note *AN204824 - F2MC-16FX Family, Setup and Debugging with EuroScope*.

3.10 Start-Up Code

The Start-Up Code begins with two consecutive *NOP* instructions. The first one is targeted by the Reset Vector, the second one is specified by the `.END` instruction at the very end of the file. This second address is used by the emulator in case of a restart. Although both start-up addresses end up in the same code, the user can examine the address to find out whether a reset or restart (i.e. without resetting RAM and peripherals) was executed.

Then the processor status is initialized by disabling all interrupts, setting the interrupt level mask and initializing the Register Bank Pointer.

3.10.1 Set Clock Ratio

This paragraph initializes the clock. Since clock setup is rather complex, please refer to Application Note *300225 "Clocks"* for details on this topic.

3.10.2 Set External Bus Configuration

This paragraph initializes the external bus. Since external bus setup is rather complex, please refer to Application Note *300208 "16FX External Bus Interface"* for details on this topic.

3.10.3 Prepare stacks and set the default stack type

This paragraph initializes the stacks. User Stack is initialized by setting *USB* and *SP* while *CCR* is set to 0xDF. System Stack is initialized by setting *SSB* and *SP* while *CCR* is set to 0x20. *SP* is filled twice with the 16-bit near address of *__userstack_top* respectively *__systemstack_top*. This way both *RET* and *RETP* operations will not exceed the stack.

The stack areas are then filled with *STACK_PATTERN*.

3.10.4 Copy initial values to data areas

This paragraph evaluates section *DTRANS* as described in *5.2 Special Sections DCLEAR and DTRANS*. The given areas are copied.

3.10.5 Clear uninitialized data areas to zero

This paragraph evaluates section *DCLEAR* as described in *5.2 Special Sections DCLEAR and DTRANS*. The given areas are filled with zero.

3.10.6 Set Data Bank Register and Direct Bank Register

The register *DTB* is filled with the Bank Address of section *DATA*. The register *DPB* is filled with the Page address of symbol *DIRDATA_S*, which is located in section *DIRDATA*.

3.10.7 ICU register initialization workaround

After UART Scanning on some devices the Boot ROM leaves some ICU registers uninitialised. These registers have to be initialized to 0. The affected devices and the corresponding registers can be found in table Table 8.

Table 8. ICU Register Initialization

Devices affected	Registers that are not cleared
MB96326RxA, MB96326YxA, MB96356RxA, MB96356YxA	TCCSL2, TCCSL3, ICE67, ICE89, ICE1011
MB96338RxA, MB96346RxA, MB96346RxB, MB96346YxA, MB96346YxB, MB96346AxA, MB96346AxB, MB96347RxA, MB96347RxB, MB96347YxA, MB96347YxB, MB96347AxA, MB96347AxB, MB96348HxA, MB96348HxB, MB96348HxC, MB96348TxA, MB96348TxB, MB96348TxC, MB96348RxA, MB96348RxB, MB96348YxA, MB96348YxB, MB96348AxA, MB96348AxB, MB96348CxA, MB96348CxC, MB96384RxA, MB96384YxA, MB96385RxA, MB96385YxA, MB96386RxA, MB96386RxB, MB96386YxA, MB96386YxB, MB96387RxA, MB96387RxB, MB96387YxA, MB96387YxB	ICE01, ICE67

3.10.8 Wait for clocks to stabilize

CKMR:6 indicates whether clock stabilization time has expired. If for any case the Start-Up File must wait for clock stabilization, it polls *CKMR:6*.

3.10.9 Initialize Low-Level Library Interface

If configured, void function *__stream_init* is called without parameters. The function is imported from the standard C Library.

3.10.10 Call C-language main function

Void function *_main* is called without parameters. The function must be defined by the application.

3.10.11 Shut down library

If configured, void function *_exit* is called without parameters. The function is imported from the standard C Library.

3.10.12 Program end loop

After returning from *_main* the CPU will be caught in an endless loop.

4 Memory Models and Addressing Schemes

This chapter explains the memory model and data section mechanism of the *F2MC-16 Family Softune™ C Compiler*, which have to be initialized by the Start-Up File.

All F2MC 16FX-microcontrollers use a 24-bit addressing scheme. 24-bit addresses are called far addresses.

In order to safe execution time and code size the address space is divided into *Banks*. Each Bank has a size of 64 kByte allowing 16-bit addressing within the Bank, which can be handled faster, since it matches the controller's word width of 16 bits. Such addresses are called near addresses. The Bank is then determined by a *Bank Register*.

There is also an 8-bit addressing scheme, called direct addressing that divides each Bank into 256 Pages. This addressing scheme is mainly used for special IO registers.

For details on memory models and addressing schemes that go beyond the scope of this document, please refer to *F2MC-16 Family Softune™ C Compiler Manual* (FccE.pdf) and *F2MC-16 Family Softune™ Assembler Manual* (FasmE.pdf) that both come with your *Softune* installation.

4.1 Far-Addressing

24-bit far addresses can be forced by usage of the *__far* qualifier.

Example: The symbol Var has to be loaded with 0x1234.

```
__far unsigned int Var;          // global variable

__far void foo(void)             // function definition
{
   Var = 0x1234;                 // value assignment
}
```

Variable *Var* is linked to section *FAR_DATA* and function *foo* is linked to section *FAR_CODE*. Note that both *Var* and *foo* are accessible independently of link target and Bank Register settings.

Depending on optimization level the compiler could produce the following code:

```
4B53030000     MOVL A, #_Var          ; move full address to accumulator
71A0           MOVL RL0, A     ; move full address to general purpose
                                  ; register
4A3412         MOVW A, #0x1234        ; move new value to accumulator
6F3800         MOVW @RL0+00, A        ; move new value to Var
```

Following code has the same result but uses a temporary Bank Register:

```
4200           MOV A, #bnksym _Var ; move bank number of Var to accumulator
6F11           MOV ADB, A           ; Additional Data Bank register := bank
                                   ; of Var
06             ADB                   ; next instruction uses ADB as bank
                                   ; pointer
73DF53033400 MOVW _Var, #0x1234   ; address (ADB<<16 + offset(Var)) := new
                                   ; value
```

Call to and return from a far addressed function produces the following code:

```
657856FF            CALLP _foo    ; call function by full address
...
              __foo: …            ; do anything
66                  RETP          ; return by 24 Bit return address
```

4.2 Near-Addressing

16-bit near addresses are expanded to 24-bit addresses by prefixing the value of the appropriate Bank Register.

Table 9. Bank Registers

Bank Register	Use case
Data Bank **DTB** Additional Data Bank **ADB**	Global and static local variables, including constants
System Stack Bank **SSB** User Stack Bank **USB**	Local variables, function parameters on stack
Program Counter Bank **PCB**	Function calls and returns from function

Near addresses can be forced by usage of the *__near* qualifier.

Example:

```
__near unsigned int Var;   // global variable

__near void foo(void)              // function definition
{
   Var = 0x1234;                   // value assignment
}
```

Variable *Var* is linked to section *DATA* and function *foo* is linked to section *CODE*. Note that Bank Registers must be set appropriately before any access to *Var* or *foo*. The compiler will not generate such code automatically. In case all variables and all code each fit in a single Bank, the example saves code size and execution time compared to the far addressed example. The Start-Up File will initialize the appropriate Bank Registers to static values.

Depending on optimization level the compiler could produce the following code:

```
73DF53033412 MOVW _Var, #0x1234  ; move new value to variable
```

Call to and return from a far addressed function produces the following code:

```
647856        CALL _foo             ; call function by offset
...
      __foo:
...                                 ; do anything
67            RET                 ; return by offset
```

Note: It depends on the instruction, which dedicated register is used as default Bank. The compiler takes care of this and utilizes this for common data, function frame, stack data and code related data (e.g. switch()-jump tables).

4.3 Direct addressing

8-bit direct addresses are expanded to 24-bit addresses by prefixing the value of the data Bank Register and direct Page register.

Table 10. Direct Address Registers

Direct Address Register	Purpose
Data Bank **DTB**	8 upper address bits
Direct Page **DPR**	8 middle address bits

Direct addresses can be forced by usage of the *__direct* qualifier. It can only be applied to global variables and constants.

Example:

```
__direct unsigned int Var; // global variable

void foo(void)                    // function definition
{
   Var = 0x12;                    // value assignment
}
```

Variable *Var* is linked to section *DIRDATA*. Note that DTB and DPR must be set appropriately before any access to *Var* or *foo*. The compiler will not generate such code automatically. The Start-Up File will initialize the appropriate Bank and Page Registers to static values.

Depending on optimization level the compiler could produce the following code:

```
445312        MOV S:_Var, #0x12   ; move new value to variable
```

4.4 Default Memory Models

To make the *F2MC-16 Family Softune™ C Compiler* capable of compiling standard C-Code that does not contain *__near* and *__far* qualifiers, a default addressing scheme is applied to all non-qualified Symbols. This default setting is called the *Memory Model*.

The *Memory Model* defines a separate default address scheme for data[2] symbols and code[3] symbols resulting in four *Memory Models*.

Table 11. Memory Models

Memory Model	Default address for data	Default address for code
Small	16 bit (__near)	16 bit (__near)
Medium	16 bit (__near)	24 bit (__far)
Compact	24 bit (__far)	16 bit (__near)
Large	24 bit (__far)	24 bit (__far)

[2] Addresses of variables and constants as well as storage size of pointer variables to variables or constants.

[3] Addresses of functions as well as storage size of pointer variables to functions.

It is always possible to override the default address scheme by explicitly specifying a size qualifier. E.g. Even if small model is used, a variable can have an address of 24-bit in size by qualifying it explicitly as *__far*.

Note: When calling a *__near* type qualified function the calling function must be positioned in the same Bank. The reason is that the corresponding Bank Register is not updated when a near function is called.

E.g. If in Medium model a function is declared as *__near*, it has to be located in the same Bank as the calling function.

4.4.1 Small Model

The small model is to be specified in situations where all code and data can be positioned within a 16-bit address space each. Since all addresses are expressed using 16 bits, a compact, high-speed program can be released.

4.4.2 Medium Model

The medium model is to be specified in situations where code can be positioned in a 24-bit address space and data can be positioned in a 16-bit address space.

It is the preferred model for all single-chip applications with more than 64 kByte ROM.

4.4.3 Compact Model

The compact model is to be specified in situations where codes can be positioned in a 16-bit address space and data can be positioned in a 24-bit address space.

It mainly appears to be used for applications with external bus interface and additional RAM.

4.4.4 Large Model

The large model is to be specified in situations where all codes and data can be positioned in a 24-bit address space. Since all addresses are expressed using 24 bits, the codes used are redundant as compared to those for the small model.

It mainly appears to be used for applications of more than 64 kByte ROM and external bus interface and additional RAM.

5 Default Memory Sections

This chapter describes default sections that are generated by the compiler in order to allow a Start-Up script to initialize certain basic settings appropriately.

The ANSI specification requires global data to be initialized. This has to be done by the Start-Up Code, since RAM contents have to be modified. However, the compiler has to provide the data that has to be cleared (set to zero) as well as the data that has to be pre-loaded with an initial value to the start-up routine. *F2MC-16 Family Softune™ C Compiler* does that by creating default sections, which contain dedicated groups of data. A section is a unit that can be handled by the linker. It either contains initialized data or reserved areas. All sections used in an application can be found in the linkage map after linking project.

5.1 Data and Code Sections

Table 12. Default Sections

Name	Type	Location	Start Value	Purpose
DATA_module	DATA	RAM	All zero	Global far data defined in module with name *module*. Note: *DEFSECT* for *pragma section* and *pragma segment* is FAR_DATA
DATA	DATA			Global near data of all modules
DIRDATA	DIR			Direct data of all modules
LIBDATA	DATA			Data of C-library
INIT_module	DATA		DCONST_module	Global far data defined in module with name *module*. **Note:** *DEFSECT* for *pragma section* and *pragma segment* is FAR_DATA
INIT	DATA		DCONST	Global near data of all modules
DIRINIT	DIR		DIRCONST	Direct data of all modules
LIBINIT	DATA		LIBDCONST	Data of C-library
CINIT	DATA		CONST	Global near constants of all modules. **Note:** This section is obsolete if ROM-mirror is used.
SSTACK	STACK		Pattern or undefined	System Stack
USTACK	STACK			User Stack
DCONST_module	CONST	ROM	Fixed in binary	Initial values for INIT_module
DCONST	CONST			Initial values for INIT
DIRCONST	DIRCONST			Initial values for DIRINIT
LIBDCONST	CONST			Initial values for LIBINIT
CONST	CONST			Global near constants of all modules. **Note:** If ROM-mirror is used this section is mapped to Bank 0. If ROM-mirror is not used, the contents of this section are copied to CINIT on startup.
CONST_module	CONST			Far constants of module named *module*
DCLEAR	CONST			Table of all DATA_module sections
DTRANS	CONST			Table of all INIT_module / DCONST_module sections
CODE_module	CODE			Far code of module named *module*
CODE	CODE			Near code of all modules
INTVECT	CONST			Interrupt vector table
RESVECT	CONST			Reset Vector
IOBASE	IO	IO area		Specific functions' registers

Sections **DATA**, **DIRDATA**, **INIT**, **DIRINIT**, **CINIT**, **DCONST**, **DIRCONST**, **CONST** and **CODE** are related to *__near* and *__direct* symbols. Because of 16-bit addressing the amount of data cannot exceed 64 KB. Therefore, all symbols of all modules are collected in the appropriate section.

Sections **DATA_module**, **INIT_module**, **DCONST_module**, **CONST_module** and **CODE_module** are related to *__far* symbols. Note that “module” is only a placeholder for several module names (e.g. DATA_main and DATA_uart for main.c and uart.c). Because of 24-bit addressing the entire space of all data can exceed 64 KB. However, the maximum size of a single section is 64 KB. Therefore, all far data are grouped in that way that every module has its own far section. The generic name of this section consists of the default name plus underscore plus the name of the module (file name of source file without extension).

Sections **DATA**, **DATA_module**, **DIRDATA**, **LIBDATA** have to be set to zero by the Start-Up Code.

Sections **INIT**, **INIT_name**, **DIRINIT**, **LIBINIT** and possibly **CINIT** have to be initialised with a start value. The appropriate start values are located in sections **DCONST**, **DCONST_name**, **DIRCONST**, **LIBDCONST** and possibly **CONST**. These sections are just copied. Therefore, the order of the start values within each section has to be the same as the order of the symbols. The compiler manages this.

5.2 Special Sections DCLEAR and DTRANS

All sections DATA_module, INIT_module and DCONST_module (note, “module” is only a placeholder for several modules) can be spread over the full address space. Therefore, they cannot be initialized as one block. They have to be initialized separately. Two tables are used to collect the locations of the far sections. The compiler manages to create these tables.

Section DCLEAR contains the table for DATA, DIRDATA, LIBDATA and all DATA_module sections. Each entry consists of start address (4 Bytes) and length (2 Bytes) of the respective DATA_module section. The size of DCLEAR results from the number of DATA_module sections multiplied by 6 Byte (size of one entry).

Section DTRANS contains the table for DCONST / INIT, DIRCONST / DIRINIT, LIBDCONST / LIBINIT, CONST / CINIT and for all DCONST_module / INIT_module sections. Each entry consists of start address of source (4 Bytes), start address of destination (4 Bytes) and length (2 Bytes) of the sections. The size of DTRANS results from the number of sections’ pairs multiplied by 10 Byte (size of one entry).

5.3 Sections CINIT vs. CONST and RAMCONST vs. ROMCONST

Sections CINIT and CONST are related to constant near data. RAMCONST and ROMCONST are compiler options, which also affect the Start-Up Code.

Constant data are fixed during run-time. They cannot be changed by C-statements. Therefore, constant data can be located in ROM area. Table 12 shows that the sections CONST and CONST_module are located in ROM. This is straightforward for CONST_module. Because these sections are far addressed and can be accessed where ever they are.

Section CONST is near addressed. This stays in conflict to the fact that the default near Bank is usually not located in ROM but in Bank 0. There are two alternatives to solve this problem.

5.3.1 RAMCONST

RAMCONST means that all near constants have to be copied to the default near Bank. In this case section CONST is copied to section CINIT. During run-time the compiler accesses all data in CINIT. The advantage is that this is independent of the location of CONST. The disadvantage is that RAM is occupied by CINIT.

Especially in single-chip applications RAM should not be used for data, which will never change.

5.3.2 ROMCONST

ROMCONST means that all near constants are mirrored to the default near Bank by hardware. In this case section CONST is accessed in ROM. Section CINIT is not used. During run-time the compiler accesses all data by the 16-bit offset in CONST. Therefore, CONST has to be linked to the region that is mirrored. The advantage is that no RAM is wasted. The disadvantage is that location of CONST is limited to the mirrored area and that the size of CONST is limited by the size of the mirrored area.

ROMCONST is the preferred setting for single-chip applications.

F²MC16FX controller can mirror a ROM area of 8, 16, 24 or 32 kByte in size from one of Banks F0..FF to Bank 0. The mirrored area is located at the very end of both Banks.

E.g. If ROM mirror is configured to mirror 24 kByte from Bank F8, then any attempt to read addresses H’00A000..H’00FFFF will return a value stored at H’F8A000..H’F8FFFF.

The Compiler generates code that accesses these constants by near addressing the same way as near addressed non-constant variables are accessed.

6 Related Application Notes

Overview of Related Application Notes and Contact to Cypress

6.1 Manuals and Application Notes

- AN205547: F2MC-16FX Family, Emulator System MB2198-01 Installation Guide
  Installing the 16FX emulation system
- AN205555: F2MC-16FX Family, Emulator System MB2198
  How to use the emulator MB2198-01 together with the Softune Workbench
- AN204824: F2MC-16FX Family, Setup and Debugging with EuroScope

Document History

Document Title: AN209372 - F2MC-16FX Family 16-Bit Microcontroller All Series Start-up File

Document Number: 002-09372

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	-	03/17/2008	V 1.0: First version; RLA.
			10/19/2009	V 1.1: Clarified relation of stack bank and compiler behavior; RLA.
*A	6091180	NOFL	03/07/2018	Updated Fujitsu Application Note “MCU-AN-300241-E-V11” to Cypress template. Completing Sunset Review.

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 MCU

Cypress Developer Community

Community | 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, 2008-2018. 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. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. 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.