



Application Note

FCC907S-Memory Initialisation with F²MC16-Start-Up File

© Fujitsu Microelectronics Europe GmbH, Microcontroller Application Group

CONTENTS

0	Introduction.....	4	2.3	CONSTDATA - Constant Symbol Area.....	14
1	FCC907S-Compiler Memory Models	4	2.4	STACKUSE - Used Stack Type	14
1.1	Memory Models	4	2.5	SSSIZE, USSIZE - Stack Size	14
1.2	Addressing Space in Detail	5	2.6	REGBANK - Register Bank.....	15
1.2.1	__far – Physical Addressing.	5	2.7	CLIBINI – Initialise C-Library.....	15
1.2.2	__near – Offset Addressing .	6	2.8	CLOCKSPEED, CLOCKWAIT – Operation Frequency	15
1.2.3	__direct – Offset Addressing	7	2.9	BUSMODE – External Bus Interface	16
1.3	FCC907S Default Data Sections.....	7	2.10	ROMMIRROR (F ² MC-16LX only)...	16
1.3.1	Section Overview.....	7	2.11	External Bus Setting	16
1.3.2	Special Sections DCLEAR and DTRANS	8	2.12	RESET_VECTOR.....	17
1.3.3	Sections CINIT vs. CONST and RAMCONST vs. ROMCONST ..	9	3	Appendix	18
1.3.4	Examples for Data Sections.	9	3.1	Source Code of Start.asm.....	18
2	Configuration of Start-up File	13	3.2	History	26
2.1	FAMILY - Controller Family.....	13			
2.2	MODEL - Memory Model	13			

Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH restricts its warranties and its liability for software (like this start-up file, other examples and tools), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling.

1. Fujitsu Microelectronics Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials (this application note) for a period of 90 days from the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Microelectronics Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.

2. Should a Product turn out to be defect, Fujitsu Microelectronics Europe GmbH's entire liability and the customer's exclusive remedy shall be, at Fujitsu Microelectronics Europe GmbH's sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Microelectronics Europe GmbH in original packing and without further defects resulting from the customer's use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Microelectronics Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Microelectronics Europe GmbH.

3. To the maximum extent permitted by applicable law Fujitsu Microelectronics Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.

4. To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH's and its suppliers' liability is restricted to intention and gross negligence.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES

To the maximum extent permitted by applicable law, in no event shall Fujitsu Microelectronics Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect.

0 Introduction

All C-language applications need special code to be executed before all others, which initialises all data areas generated by the compiler and which performs a 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.am” version 1.9 provided by Fujitsu Microelectronics Europe GmbH for all F²MC16-microcontrollers. The start-up file “start.asm” comes with the Softune Workbench FME’s edition and can be found in the Sample directory. Please also refer to the Softune Workbench help.

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 initialisation according to FCC907S-compiler sections,
- Start-up file is linkage order independent,
- Start-up file is suitable for all memory models.

1 FCC907S-Compiler Memory Models

In order to understand the purpose of the data areas, which have to be initialised by the start-up file, this chapter explains the memory model and data section mechanism of the FCC907S-compiler. The compiler help provides additional information. It is also advisable to check the list files (*.lst) generated by compiler/assembler for better understanding of what compiler is doing with symbols and sections. Linkage information files (*.mp1) of the project should also be checked after linkage.

1.1 Memory Models

The memory model is a compiler option for simplifying the usage of 16 Bit- and 32 Bit-addressing. The memory model defines the default type of addressing. By changing memory model the access type of all symbols changes automatically.

Model	Default data address	Default code address
Small	16 Bit (<code>__near</code>)	16 Bit (<code>__near</code>)
Medium	16 Bit (<code>__near</code>)	24 Bit (<code>__far</code>)
Compact	24 Bit (<code>__far</code>)	16 Bit (<code>__near</code>)
Large	24 Bit (<code>__far</code>)	24 Bit (<code>__far</code>)

Table 1 Memory model, address space and appropriate C-qualifier

Note, 16 Bit addressing by memory model setting implies the usage of `__near` qualifier, even if not written in source-code. Note, 24 Bit addressing by memory model setting implies the usage of `__far` qualifier, even if not written in source-code.

It is always possible to overwrite the default address size of the memory model by dedicatedly specifying the size qualifier `__near` or `__far`.

E.g. Even if Small model is used, a variable can have 24 Bit address space by specifying `__far`.

Note: When calling a `__near` type qualified function from a `__far` type qualified function, both functions must be positioned in the same bank. The reason is that the PCB set up for `__far` type qualified function calling is also used for `__near` type qualified function calling. E.g. If in model Medium a function is declared as `__near`, it has to be located in the same bank as the calling function.

Small Model

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

Medium Model

The medium model is to be specified in situations where codes 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 KB ROM.

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 with applications with external bus interface and additional RAM.

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 with applications with external bus interface and additional RAM.

1.2 Addressing Space in Detail

For programming in C the FCC907S-compiler provides the qualifiers “`__far`”, “`__near`” and “`__direct`”. These describe whether a symbol is addressed by its full, physical address, by 16 Bit offset or by lowest 8 Bit only. Using different addressing results in different instructions to be used. Therefore, `__direct`, `__near` and `__far` addressing can be used in order to achieve improvement of performance (`__direct`, `__near`) or flexibility (`__near`, `__far`).

Qualifier	<code>__direct</code>	<code>__near</code>	<code>__far</code>
Address size	8 Bit	16 Bit	24 Bit
Address space	256 Byte	64 KByte	16 MByte

Table 2 Address qualifiers and address size relation

1.2.1 `__far` – Physical Addressing

Physical addressing means that the full controller address space is covered. F²MC16-microcontrollers have a 24 Bit address size, which results in a 16 MB address space. Specifying `__far` to a symbol (qualifier in C-source code or appropriate memory model) forces physical addressing. The advantage is that the symbol is accessible regardless of the address of the symbol. Disadvantage is that always 24 bits have to be handled, which requires more code and reduces performance.

E.g. the symbol `Var` has to be loaded with 0x1234. It has been located in bank 0 at address H'0000353 by the linker.

```
unsigned int __far Var;           /* global variable */
void foo(void) { Var = 0x1234;}  /* move new value to Var */
```

Example 1 Usage of `__far` qualifier in C forces physical addressing of `Var`

Following assembler instructions might be used:

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

Example 2 Physical addressing via general purpose register

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

Example 3 Physical addressing via bank register

Physical addressing for code looks like this:

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

Example 4 Physical addressing of the function void __far foo(void) at 0xFF5678;

1.2.2 __near – Offset Addressing

Offset addressing means that only lower 16 address bits are handled during run-time. Advantages are smaller code size and faster execution time. Disadvantage is the limited address range of 64 KB.

Even if the symbol is accessed by 16 Bit address only, it still has a physical address it has been linked to by the linker. When using __near addressing (by qualifier in source code or using appropriate memory model) the missing eight bits of the physical address are fixed during run-time as a global setting. The compiler uses following dedicated registers as fixed bank addresses in order to resolve the physical address for __near addressed symbols:

Data Bank DTB	global and static local variables
User Stack Bank USB, System Stack Bank SSB	local variables, function parameters on stack
Program Counter Bank PCB	function calls and returns from function

Table 3 Symbol types and default bank register

It is the task of the start-up file to provide the correct setting of those registers before entering application *main()* function. Usually bank 0 (containing internal RAM) is the default __near bank. Therefore, DTB, USB and SSB are to be set to 0 in most cases.

Using examples of 1.2.1 __far – Physical Addressing changes as follows:

unsigned int __near Var;	/* global variable */
void foo(void) { Var = 0x1234;}	/* move new value to Var */

Example 5 Usage of __near qualifier in C forces offset addressing of Var

Following assembler instructions might be used:

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

Example 6 Only one instruction is necessary to load Var

Offset addressing for code looks like this:

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

Example 7 Offset addressing of the function void __near foo(void);

Note: It depends on the instruction, which dedicated register is used as default bank. The compiler takes care of this and utilises this for common data, function frame, stack data and

code related data (e.g. *switch()*-jump tables). Please refer to the compiler help and programming manual for more information on this.

1.2.3 `__direct` – Offset Addressing

Direct addressing means that only lowest 8 address bits are handled during run-time. Advantages are very small code size and faster execution time. Disadvantage is the limited address range of 256 Byte and the limited number of instruction for direct addressing. It can only be assigned to global variables and constants but not to functions. All `__direct` variables must be located within one page (upper 16 address bits must be same). The `__direct` page must be located within the default `__near` bank.

Even if the variable is accessed by 8 Bit address only, it still has a physical address it has been linked to by the linker. When using `__direct` addressing the missing sixteen bits of the physical address are fixed during run-time as a global setting. The compiler uses following dedicated registers as bank and page registers in order to resolve the physical address for `__direct` addressed symbols:

DTB	Upper eight address bits of <code>__direct</code> variables
DPR	Middle eight address bits of <code>__direct</code> variables

Table 4 Default registers for upper 16 address bits of `__direct` variables

It is the task of the start-up file to provide the correct setting of those registers before entering application *main()* function. If correct sections and section types (see 1.3) and appropriate linker options are used, the linker takes care of the correct page alignment.

Examples of 1.2.2 changes as follows:

```
unsigned char __direct Var;          /* global variable */
void foo(void) { Var = 0x12;} /* move new value to Var */
```

Example 8 Usage of `__near` qualifier in C forces offset addressing of *Var*

Following assembler instructions might be used:

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

Example 9 Only one instruction is necessary to load *Var*

Note: It depends on the source code and data size whether direct addressing can be used effectively. If *Var* is a word variable as in 1.2.2, compiler would automatically use `__near` addressing in function *foo()*. This is due to the fact that there is no `MOVW S:dir,#imm8-` instruction. Therefore, `__direct` addressing is most effective for byte data (also within structures). There is still a performance win with word data. However, if double word data are specified as `__direct`, one cannot expect any improvement of performance.

1.3 FCC907S Default Data Sections

1.3.1 Section Overview

The ANSI specification requires that global data are initialised. This has to be done by the start-up code. However, the compiler has to provide a mechanism to collect those data, which have to be cleared (set to zero) and those data, which have to be pre-loaded with a value. FCC907S provides default sections, which contain dedicated groups of data. A section is a unit that can be handled by the linker. It either contains initialised data or reserved areas. All sections used in an application can be found in the linkage map after linking project.

Name	Type	Start value		Purpose	
DATA	DATA	RAM	Cleared	__near data of all modules	
DATA_name	DATA			All zero	__far data of module "name"
DIRDATA	DIR				__direct data of all modules
LIBDATA	DATA				All data of C-library
INIT	DATA	RAM	Initialised	DCONST	
INIT_name	DATA			DCONST_name	__far data of module "name"
DIRINIT	DIR			DIRCONST	__direct data of all modules
LIBINIT	DATA			LIBDCONST	All data of C-library
CINIT	DATA			CONST	__near constants of all modules
DCONST	CONST	ROM	Fixed in binary	Start-up data for INIT	
DCONST_name	CONST			Start-up data for INIT_name	
DIRCONST	DIRCONST			Start-up data for DIRINIT	
LIBDCONST	CONST			Start-up data for LIBINIT	
CONST	CONST			__near constants of all modules	
CONST_name	CONST			__far constants of module "name"	
DCLEAR	CONST			Table of all DATA_name sections	
DTRANS	CONST			Table of all INIT_/DCONST_name	
CODE	CODE			__near code of all modules	
CODE_name	CODE			__far code of modules "name"	
INTVECT	CONST			Interrupt vector table	
IO	IO	Controller IO area		Specific functions registers	

Table 5 Default section types of FCC907S compiler. Note that "name" is only a placeholder for several module names.

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_name**, **INIT_name**, **DCONST_name**, **CONST_name** and **CODE_name** are related to `__far` symbols. Note that "name" 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` sections. 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_name**, **DIRDATA**, **LIBDATA** have to be set to zero by the start-up code.

Sections **INIT**, **INIT_name**, **DIRINIT**, **LIBINIT** and possibly **CINIT** (see 1.3.3) have to be initialised with a start value. The appropriate start values are located in sections **DCONST**, **DCONST_name**, **DIRCONST**, **LIBDCONST** and possibly **CONST** (see 1.3.3). 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.

1.3.2 Special Sections DCLEAR and DTRANS

All sections **DATA_name**, **INIT_name** and **DCONST_name** (note, "name" is only a placeholder for several modules) can be spread over the full address space. Therefore, they cannot be initialised as one block. They have to be initialised 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 all DATA_name sections. One entry consists of start address (4 bytes) and length (2 bytes) of the respective DATA_name section. The size of DCLEAR results from the number of DATA_name sections multiplied by 6 Byte (size of one entry).

Section DTRANS contains the table for all INIT_name and DCONST_name sections. One entry consists of start address of DCONST_name (4 bytes), start address of INIT_name (4 bytes) and length (2 bytes) of INIT_name/CONST_name sections. The size of DTRANS results from the number of INIT_name/DCONST_name sections multiplied by 10 Byte (size of one entry).

1.3.3 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 2 shows that the sections CONST and CONST_name are located in ROM. This is straightforward for CONST_name. Because these sections are `__far` addressed and can be accessed where ever they are.

Section CONST is `__near` addressed. That conflicts with the fact that the default `__near` bank is usually not located in ROM but in bank 0. There are two alternatives to overcome this.

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.

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.

E.g. F²MC16 controller have the ROM area H'FF4000...H'FFFFFF (48 KB) mirrored to H'004000...H'00FFFF. The constant variable `const int __near Var = 100` is located in CONST at address H'FF4567. The Compiler uses offset and generates code that accesses Var at address H'004567 by bank register (DTB=0) and offset H'4567.

1.3.4 Examples for Data Sections

Following table lists the used section for most often used symbol types. All case not listed here can be examined by simply compiling C-source file with single definition statement only. The list file (*.lst) can be checked for occurrence of "section" statement.

This table assumes the source file test.c to be used. "Descriptor" of DCLEAR and DTRANS means the table entry (see 1.3.2). Note that even if more than one variable using DCLEAR/DTRANS is defined in a module, the descriptor is generated only once per module.

C-source	RAM		ROM	
	Section and Contents	Size	Section and Contents	Size
Simple data and fields				
<code>__near int var; __near char field[2];</code>	DATA gets label	2		
<code>__near int var = 1; __near char field[2] = {1, 0}; __near char field[] = „\001“; // (01 00)</code>	INIT gets label	2	DCONST gets start value (0001)	2
<code>__near const int var = 2; __near const char field[] = {2,0};</code>	ROMCONST		CONST gets label and start value (0002) Data are accessed in ROM via ROM mirror	2
	RAMCONST	CINIT gets label	CONST gets start value (0002), has to be copied to CINIT	2
<code>__far int var; __far char field[2];</code>	DATA_test gets label	2	DCLEAR gets descriptor of DATA_test	6
<code>__far int var = 3; __far char field[2] = {1, 0}; __far char field[] = „\001“; // (01 00)</code>	INIT_test gets label	2	DCONST_name gets start value (0003) DTRANS gets descriptor of INIT_test	2 10
<code>__far const int var = 4; __far const char field[] = {2,0};</code>			CONST_test gets label and value (0004)	2
<code>__direct int var; __direct char field[2];</code>	DIRDATA gets label	2		
<code>__direct int var = 5; __direct const int var = 5; __direct char field[2] = {5, 0}; __direct const char field[] = {5,0};</code>	DIRINIT gets label	2	DIRCONST gets start value (0005)	2
Simple pointers				
<code>__near void * __near ptr;</code>	DATA gets label	2		
<code>__near void * __near iptr = (void*)7;</code>	INIT gets label	2	DCONST gets start value (0007)	2
<code>__near char * __near ptr = „str1“;</code>	ROMCONST	INIT gets label “ptr” and is initialised with addr of hidden label in DCONST	CONST gets hidden label and value “str1” DCONST gets reference to hidden label	5 2
	RAMCONST	CINIT gets hidden label and is initialised from CONST	CONST gets start value “str1”	5
		INIT gets label “ptr” and is initialised with addr of hidden label in DCONST	DCONST gets address of hidden label in CINIT	2
<code>__near void * __far ptr;</code>	DATA_test gets label	2	DCLEAR gets descriptor of DATA_test	6
<code>__far void * __near ptr;</code>	DATA gets label	4		

Table 6 Variables definitions in C and used sections

2 Configuration of Start-up File

The start-up file has to be configured for the target system (hardware) and the application software. The start-up file provides several prepared settings, which have to be checked and which have possibly to be changed. The header of the start-up file contains information how to find the lines containing the options to be checked. With version 1.9 of the start-up file all lines with the comment extension “; <<<” were pointing to available settings.

In the following paragraphs all available settings are described.

2.1 FAMILY - Controller Family

Set family type of the chip the application isv running in. This setting is used by the start-up file to check the availability of registers and to check the plausibility of settings. The correct family setting is found by simplifying the name of the implemented controller.

Available settings:

- MB90700 → F²MC16H-family
- MB90200 → F²MC16F-family
- MB90600 → F²MC16L-family
- MB90500 → F²MC16LX-family
- MB90400 → F²MC16LX-family

Note: With this start-up file version only LX-families are differed from non-LX-families.

Note: MB90500 and MB90400 are both LX-families. There are minor core changes.

Note: This start-up file version has not been tested with MB90700 and MB90200.

E.g.: Used controller MB90F598 → set MB90500 family, LX-family

2.2 MODEL - Memory Model

The memory model describes how data and code are accessed by compiler-generated assembler code. Please refer to the chapter *1 FCC907S-Compiler Memory Models* in order to understand the mechanism. Available options are SMALL, MEDIUM, COMPACT and LARGE.

Within this 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 is still mandatory.

The setting AUTO 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 and not with RETP-instruction. For 64 KB code-models the bank information on stack can be ignored. The remaining two bytes on stack (CALLP pushes four bytes onto stack) are automatically removed by the start-up code. Therefore, the AUTO-setting is working with all memory models.

The start-up file setting MODEL does not affect data accesses and data initialisations. Regardless of the memory model all section types are always initialised (except CINIT, see 1.3.3).

If AUTO setting is used, the start-up file has not to be changed or modified, if the compiler memory model is changed. AUTO is the recommended setting.

2.3 CONSTDATA - Constant Symbol Area

The CONSTDATA setting specifies how 16 Bit-addressed (`__near`) constant data are handled by the start-up file. Please also refer to the chapter *1 FCC907S-Compiler Memory Models* in order to understand the mechanism.

The start-up file setting CONSTDATA only controls whether the constants are actually copied to from CONST (ROM) to CINIT (RAM) by start-up code. The setting does not affect the compiler option RAMCONST. To set the compiler to the correct mode is still mandatory.

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 initialise 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.

Recommended start-up file option is RAMCONST because this works regardless of the compiler setting. Preferred compiler option is ROMCONST because no RAM is used for constant data. 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. However, since the compiler will not generate data for the RAMCONST section (CINIT), the byte count is zero and no data are copied.

Note that the ROMMIRROR (see 2.10) has to be enabled in all internal ROM modes, if compiler is set to ROMCONST.

2.4 STACKUSE - Used Stack Type

The setting STACKUSE specifies the stack type that is configured, when the application *main()*-function is called.

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.

System stack 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 (option SYSSTACK).

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.

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.

2.5 SSSIZE, USSIZE - Stack Size

These settings specify the amount of words (16 Bit) to be reserved for stacks. This value has to cover all:

- parameters passed over stack

- return addresses for function calls
- local variables (except static)
- temporary data due to compiler optimisation
- interrupt context on stacks
- 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 Workbench installation.

If the support tool “C-Analyzer” is purchased, the additional tool “Musc” is able to collect these data and to calculate the minimum stack size for the whole application.

Note: If STACKUSE is set to SYSSTACK, the number of words for the user stack is automatically limited to one word by the start-up file.

2.6 REGBANK - Register Bank

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

F²MC16-microcontroller 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.

2.7 CLIBINI – Initialise C-Library

The setting CLIBINI specifies whether the start-up code has to call the stream initialisation function of the C-library.

The stream initialisation 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 more information refer to the compiler help.

2.8 CLOCKSPEED, CLOCKWAIT – Operation Frequency

The setting CLOCKSPEED specifies the operation frequency to be set by the start-up code. CLOCKWAIT specifies whether the PLL has to be stable before calling application *main()*-function.

With this settings either main clock (PLL off) or any PLL-mode can be selected. If the clock setting has to be handled by the application main code, the option NOCLOCK avoids any access to the clock control register by the start-up code.

If any PLL-mode has been selected by CLOCKSPEED, the setting CLOCKWAIT specifies whether to wait for stabilised PLL before calling *main()*-function. If the start-up code selects a PLL-mode, it is not immediately activated. The controller still runs at main-clock mode until the PLL-stabilisation wait time runs out. After that the controller switches automatically to the selected PLL-mode. If *main()*-function is called when controller is still in main-clock mode, the usage of resources might cause misoperation (wrong timing). To avoid this CLOCKWAIT should be set. This way the start-up code waits until the PLL is activated and delays the call of the *main()*-function.

2.9 BUSMODE – External Bus Interface

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 INTROM_EXTBUS, the internal ROM (Mask, Flash, OTP) 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.

- BUSMODE affects the mode byte setting, if the reset vector is enabled in the start-up file (see 2.12).
- If INTROM_EXTBUS or EXTROM_EXTBUS is selected, external bus control registers are set by the start-up code (see 2.11). In SINGLE_CHIP mode external bus control registers are not accessed.
- If INTROM_EXTBUS or EXTROM_EXTBUS is selected, external bus control registers are set by the start-up code (see 2.11). In SINGLE_CHIP mode external bus control registers are not accessed.

2.10 ROMMIRROR (F²MC-16LX only)

This setting specifies the usage of the ROM-mirror function for of F²MC-16LX-microcontrollers in INTROM_EXTBUS mode. F²MC-16LX controllers have a special function register that allows choosing whether to access internal ROM or external bus in the area H'004000...H'00FFFF.

If ROMIRROR is set to ON, the internal ROM area H'FF4000...H'FFFFFF is mirrored to H'004000...H'00FFFF. If ROMIRROR is set to OFF, the external bus is visible in H'004000...H'00FFFF.

This setting does not apply to other controllers than F²MC-16LX. It does not apply to other bus modes than INTROM_EXTBUS. For other controller families and other bus modes it is ignored. In single-chip mode of F²MC-16LX controllers and in internal-ROM modes of other controllers the ROM area is always mirrored.

Recommended setting is ON because it allows the usage of the ROMCONST-option of the compiler in internal-ROM modes (see 1.3.3).

2.11 External Bus Setting

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 hardware controller hardware manual for detailed information.

AUTOWAIT_IO, AUTOWAIT_LO, AUTOWAIT_HI

These settings refer to the Auto-Ready function (wait-states). No wait-states, one, two or three wait-states can be set separately for the external IO-area (C0...FF), the lower external area (002000...7FFFFFFF) and to the higher external area (800000...FFFFFFF/800000...end of external bus area).

ADDR_PINS

This setting specifies the usage of address lines A16...A23. These can be set as IO-port instead of address line independently. This allows to save IO-ports, if the external address decoder does not need to differ these address lines.

BUS_SIGNAL

This setting specifies the bus width and the usage bus control signals Write, Hold Request, Ready and Clock. The control signal pins can be used as IO-port instead of bus control signal.

This allows to save IO-ports, if not all bus features are used. E.g. If only external ROM is connected, there is not need for Write signal.

The bus width specifies to use either 16 Bit multiplexed bus or 8 Bit multiplexed bus. This can be set separately for the external IO-area (C0...FF), the lower external area (002000...7FFFFFFF) and to the higher external area (800000...FFFFFFF/800000...end of external bus area). If RESET_VECTOR (see 2.12) is enabled, the selection of the higher bus area will become part of the mode byte of the reset vector.

In 8 Bit multiplexed bus mode the address is output first. Than the AD0...AD7 changes to data bus (read or write). Upper 16 address lines (or less) keep their address value. Lower 8 address lines have to be latched. If internal memory (e.g. internal RAM) is accessed, upper 16 address lines still keep the value of the last external access. Lower 8 address lines change to high-impedance and bus control signals stay inactive (e.g. /RD stays high).

In 16 Bit multiplexed bus mode the address is output first. Than the AD0...AD15 changes to data bus (read or write). Upper 8 address lines (or less) keep their address value. Lower 16 address lines have to be latched. If internal memory (e.g. internal RAM) is accessed, upper 8 address lines still keep the value of the last external access. Lower 16 address lines change to high-impedance and bus control signals stay inactive (e.g. /RD stays high).

2.12 RESET_VECTOR

This setting specifies the generation of a reset vector. A reset vector contains the start address of the program and a mode byte, which defines the bus mode. If RESET_VECTOR is ON, a section RESVECT is generated. The mode byte is automatically calculated from the other bus settings.

If RESET_VECTOR is OFF, this section is not generated.

Note: The reset vector can also be defined by pragma statement in C-language. However, the mode byte has to be calculated manually then.

Note: For those devices, which have fixed reset vector or if bootstrap loaders are used, which do not process the vector address, RESET_VECTOR can be set OFF. In this case the start-up module has to be linked to the proper address by linker settings.

Note: If devices with fixed reset vector are used and if these devices are debugged with emulator debugger, RESET_VECTOR should be set to ON, even if the start-up code is linked to fixed start address. This is because the emulator debugger does not process fixed vectors and still needs the reset vector. Make sure that the reset vector is identical with the fixed vector, if no special bootstrap loader (for Flash memory) is used that takes care of these different vectors.

3 Appendix

3.1 Source Code of Start.asm

```

;=====
;   MB90500/MB90600/MB90700/MB90700H/MB90200 Series C Compiler,
;   (C) FUJITSU MICROELECTRONICS EUROPE GMBH 1998-99
;
;   Startup file for memory and basic controller initialization
;=====
;
;   .PROGRAM   STARTUP
;   .TITLE     "STARTUP FILE FOR MEMORY INITIALIZATION"
;
;=====
; CHECK ALL OPTIONS WHETHER THEY FIT TO THE APPLICATION;
;
; Configure this startup file in the "Settings" section. Search for
; comments with leading "; <<<". This points to the items to be set.
;
;=====
;   Disclaimer
;=====
;
;           FUJITSU MICROELECTRONICS EUROPE GMBH
;           Am Siebenstein 6-10, 63303 Dreieich
;           Tel.:+49/6103/690-0,Fax - 122
;
;   The following software is for demonstration purposes only.
;   It is not fully tested, nor validated in order to fullfill
;   its task under all circumstances. Therefore, this software
;   or any part of it must only be used in an evaluation
;   laboratory environment.
;   This software is subject to the rules of our standard
;   DISCLAIMER, that is delivered with our SW-tools (on the CD
;   "Micros Documentation & Software V3.0" see "\START.HTM" or
;   see our Internet Page -
;   http://www.fujitsu-edc.com/products/micro/disclaimer.html
;
;=====
;   History
;=====
;
; Version 1.00      25. Aug 98  Holger Loesche
; - original version
; Version 1.01      31. Aug 98  Holger Loesche
; - bug: conditional for reset vector was missing
; Version 1.02      16. Oct 98  Holger Loesche
; - memory model AUTO introduced (use far calls only and repair
;   stack, if necessary
; - colons removed from EQU labels
; - stream_init call added
; - RAMCONST set as default (also for ROMCONST systems)
; Version 1.03      19. Oct 98  Holger Loesche
; - bug: SEGCOPY macro: used size changed from sizeof(src) to
;   sizeof(dest). It was conflicting with RAMCONST, if compiler
;   is set to ROMCONST.
; Version 1.04      21. Oct 98  Holger Loesche
; - ROM mirror option added
; - _exit call added
; - bug: EQU ON/OFF move to upper lines
; Version 1.05      28. Oct 98  Holger Loesche
; - CALL/CALLP _exit was not differred
; Version 1.06      18. Feb 99  Holger Loesche
; - default external bus configuration: WR signal enabled
; - ROMMIRROR macro processing simplified (less warnings>
; Version 1.07      01. April 99  Holger Loesche
; - Version string had wrong number (1.05 instead of 1.06)
; - Copyright slightly changed
; Version 1.08      16. April 99  Juergen Rohn
; - Version placed in separate section (caused problems with fixed
;   reset vector)
; Version 1.09      12. May 99   Holger Loesche
; - MB90400 family added

```

```

; - several coments changed
; - INTRROM_EXTBUS macro was wrong
; - BUSWIDTH macro removed, no resolved from bus signal
; - disclaimer added
; Version 1.10      17. May 99      Holger Loesche
; - change in 1.10 (MODEBYTE) related to BUSWIDTH was not complete
;
;=====
        .SECTION  VERSIONS, CONST
        .SDATA    "Start 1.09\n"      ; comment this line to remove
;=====
;
;   Settings
;=====
#set     OFF      0
#set     ON       1

; ----- controller family -----

#set     MB90700  0
#set     MB90200  1
#set     MB90600  2
#set     MB90500  3
#set     MB90400  4

#set     FAMILY   MB90500      ; <<< select family

; ----- Memory models -----          default address size
;                                     ;   data   code
#set     SMALL    0           ;   16 Bit  16 Bit
#set     MEDIUM  1           ;   16 Bit  24 Bit
#set     COMPACT  2           ;   24 Bit  16 Bit
#set     LARGE    3           ;   24 Bit  24 Bit

#set     AUTO     4           ; works always, might waste 2 bytes

#set     MEMMODEL AUTO        ; <<< C-memory model

; The selected memory model should be set in order to fit to the
; model selected for the compiler.
; Note, in this startup version AUTO will work for all
; C-models. However, if the compiler is configured for SMALL or
; COMPACT, two bytes on stack will be lost. If this is not
; acceptable, the above setting should be set to the correct model.

; ----- Constant symbols -----

#set     ROMCONST 0           ; works only with compiler ROMCONST
#set     RAMCONST 1           ; works with BOTH compiler settings

#set     CONSTDATA RAMCONST   ; <<< set RAMCONST or ROMCONST

; - RAMCONST (default) should always work, even if compiler is set
;   to ROMCONST.
;   If compiler is set to ROMCONST and this startup file is set to
;   RAMCONST, this startup file will only generate an empty section
;   CINIT. The code, which copies from CONST to CINIT will not have
;   any effect then.
; - It is highly recommended to set the compiler to ROMCONST for
;   single-chip mode or internal ROM+ext bus.
; - Full external bus requires external address mapping.
;   Single-chip can be emulated by the emulator debugger.
;   ROM mirror can also be used with simulator.
;
; see also MIRROR options of external bus settings

; ----- stack -----

#set     USRSTACK 0           ; use user stack, system stack for interrupts
#set     SYSSTACK 1           ; use system stack for all (program + inter)

#set     STACKUSE SYSSTACK    ; <<< set used stacks

; - If only SSB is used and SSB is linked to a different bank than USB,
;   make sure that all C-modules (which generate far pointers to stack
;   data) have "#pragma SSB". Applies only to exclusive configurations.

```

Application Note

```
; - Note, several library functions require quite a big stack (due to
; ANSI). Check the stack information files (*.stk) in the LIB\907
; directory.

SSSIZE      .EQU 384          ; <<< system stack size in words
#if STACKUSE == USRSTACK
USSIZE      .EQU 384          ; <<< user stack size, if used
#else
USSIZE      .EQU 1            ; just a dummy
#endif

#if STACKUSE == USRSTACK
# macro RELOAD_SP              ; used after function call
    MOVW A, #USTACK_TOP        ; repair stack, if stream_init
    MOVW SP,A                  ; was completed by RET (not RETP)
# endm
#else
# macro RELOAD_SP              ; used after function call
    MOVW A, #SSTACK_TOP        ; repair stack, in case stream_init
    MOVW SP,A                  ; was completed by RET (not RETP)
# endm
#endif

; ----- General Register Bank -----

#set        REGBANK 0          ; <<< set default register bank

; set the General Register Bank that is to be used after startup.
; Usually, this is bank 0, which applies to address H'180..H'18F. Set
; in the range from 0 to 31.

#if REGBANK > 31 || REGBANK < 0
# error REGBANK setting out of range
#endif

; ----- Library interface -----

#set        NOLIBINIT 0        ; do not initialize Library
#set        DOLIBINIT 1        ; initialize Library interface

#set        CLIBINI NOLIBINIT ; <<< select extended library usage

; This option has only to be set, if file-IO/standard-IO function of the
; C-library have to be used (printf(), fopen(...)). This also requires
; low-level functions to be defined by the application software.
; For other library functions like (e.g. sprintf()) all this is not
; necessary. However, several functions consume a large amount of stack.

; ----- Clock selection -----

#set        NOCLOCK 0          ; do not touch CKSCR register
#set        MAINCLOCK 1        ; select main clock (1/2 external)
#set        PLLx1 2           ; set PLL to x1 ext. clock/quartz
#set        PLLx2 3           ; set PLL to x2 ext. clock/quartz
#set        PLLx3 4           ; set PLL to x3 ext. clock/quartz
#set        PLLx4 5           ; set PLL to x4 ext. clock/quartz

#set        CLOCKSPEED PLLx4   ; <<< set PLL ratio
#set        CLOCKWAIT ON       ; <<< wait for stabilized PLL, if
;                               ; PLL is used

; The clock is set quiet early. However, if CLOCKWAIT is ON, polling
; for machine clock to be switched to PLL is done at the end of this
; file. Therefore, the stabilization time is not wasted. Main() will
; finally start at correct speed. Resources can immediately be used.
;
; This startup file version does not support subclock.

; ----- Bus interface -----

#set        SINGLE_CHIP 0      ; all internal
#set        INTROM_EXTBUS 1     ; mask ROM, FLASH, or OTP ROM used
#set        EXTROM_EXTBUS 2     ; full external bus (INROM not used)

#set        BUSMODE SINGLE_CHIP ; <<< set bus mode (see mode pins)

#set        ROMMIRROR ON       ; <<< ROM mirror function ON/OFF
```

```

; MB90500 family only

; In Internal ROM / External Bus mode one can select whether to mirror
; area FF4000..FFFFFF to 004000..00FFFF. This is necessary to get the
; compiler ROMCONST option working. However, if ROMCONST is not used,
; this area might be used to access external memory. This is intended
; to increase performance, if a lot of dynamic data have to be accessed.
; In SMALL and MEDIUM model these data can be accessed within bank 0,
; which allows to use near addressing.
; These controller without the ROMM-control register always have the
; mirror function on in INROM mode.

; If BUSMODE is "SINGLE_CHIP", ignore remaining bus settings.

#set WIDTH_8 0 ; 8 Bit external bus
#set WIDTH_16 1 ; 16 Bit external bus

;#set BUSWIDTH WIDTH_16 ; <<< set external bus width

#set AUTOWAIT_IO 0 ; <<< 0..3 waitstates for IO area
#set AUTOWAIT_LO 0 ; <<< 0..3 for lower external area
#set AUTOWAIT_HI 0 ; <<< 0..3 for higher external area

#set ADDR_PINS B'00000000 ; <<< select used address lines
; A23..A16 to be output.
; This is the value to be set in HACR-register. "1" means: pin used as
; IO-port. (B'10000000 => A23 not used, B'00000001 => A16 not used)

#set BUS_SIGNAL B'00000100 ; <<< enable bus control signals
; | | | | | | | | _ ignored
; | | | | | | | | _ bus width lower memory (0:16, 1:8Bit)
; | | | | | | | | _ output WR signal(s) (1: enabled )
; | | | | | | | | _ bus width upper memory (0:16, 1:8Bit)
; | | | | | | | | _ bus width ext IO area (0:16, 1:8Bit)
; | | | | | | | | _ enable HRQ input (1: enabled )
; | | | | | | | | _ enable RDY input (1: enabled )
; | | | | | | | | _ output CLK signal (1:enabled )

; These settings correspond to the EPCR-register.
; Hint: Except for MB90500 devices the clock output is needed for
; external RDY synchronisation, if Ready function is used.
; Hint: Don't forget to enable WR signals, if external RAM has to be
; written to.

#set iARSR ((AUTOWAIT_IO<<6) | ((AUTOWAIT_HI&3)<<4) | (AUTOWAIT_LO&3))

; ----- Reset Vector -----

#set RESET_VECTOR ON ; <<< enable reset vector

#if BUSMODE == SINGLE_CHIP
# set MODEBYTE 0
#else
# set MODEBYTE ((BUSMODE&3)<<6) | ((~BUS_SIGNAL)&8) )
#endif

; Above setting can also be used, if all other interrupt vectors are
; specified via "pragma intvect". Only if interrupts 0..7 are specified
; via "pragma intvect", this will conflict with the vector in this
; module. The reason is the INTVECT section, which includes the whole
; area from the lowest to the highest specified vector.

#if RESET_VECTOR == ON
.SECTION RESVECT, CONST, LOCATE=H'FFFDC
.DATA.E _start
.DATA.B MODEBYTE
#endif

; <<< END OF SETTINGS >>>

;=====
; Several fixed addresses (fixed for MB90xxx controllers)
;=====

LPMCR .EQU 0xA0 ; Low power mode control register

```



```

;=====
; Program start address the reset vector should point here
;=====
_start:
    AND CCR, #0          ; disable interrupts
    MOV ILM,#7          ; set interrupt level mask to ALL
    MOV RP,#REGBANK     ; set register bank pointer

;=====
; Set clock ratio (ignore subclock)
;=====
#if CLOCKSPEED != NOCLOCK
    SETB I:CKSCR:2      ; set main clock
# if CLOCKSPEED > MAINCLOCK
    MOV A, I:CKSCR      ; copy clock register
    AND A, #0xFC       ; set x1 for PLL
#   if CLOCKSPEED == PLLx2
        OR A, #0x01     ; set x2 for PLL
#   elif CLOCKSPEED == PLLx3
        OR A, #0x02     ; set x3 for PLL
#   elif CLOCKSPEED == PLLx4
        OR A, #0x03     ; set x4 for PLL
#   endif
    MOV I:CKSCR, A      ; write back
    CLRB I:CKSCR:2     ; enable PLL, PLL is not switched
                        ; to the MCU yet but after stabi-
                        ; lizing it switches on its own to
                        ; higher speed (see below)
# endif ; CLOCKSPEED > MAINCLOCK
#endif ; CLOCKSPEED != NOCLOCK

;=====
; Set some external bus configuration
;=====
#if BUSMODE != SINGLE_CHIP ; ext bus used
    MOV I:HACR, #ADDR_PINS ; set used upper address lines
    MOV I:EPGR, #BUS_SIGNAL ; set used bus signals
    MOV I:ARSR, #iARSR     ; set auto-wait cycles
#endif

#if FAMILY == MB90500 || FAMILY == MB90400 ; only these have ROMM
# if BUSMODE == INTROM_EXTBUS ; EXTBUS and INTROM/EXTROM
#   if ROMMIRROR == OFF && CONSTDATA == ROMCONST
#     error Mirror function must be ON to mirror internal ROM
#   endif
# endif

    MOV I:ROMM, #ROMMIRROR
#endif

;=====
; Copy from initial value areas to reserved data area of near data
;=====
#macro   SEGCOPY DEST, SRC
    MOV A, #BNKSEC \SRC ; get bank of source section
    MOV DTB, A          ; store source bank in DTB
    MOV A, #BNKSEC \DEST ; get destination bank
    MOV ADB, A          ; store dest bank in ADB
    MOVW RW0, #SIZEOF (\DEST) ; get size of dest section
    MOVW A, #\DEST      ; move destination offset to AL
    MOVW A, #\SRC       ; move source offset to AL and
                        ; move AL (dest offset) to AH
    MOVSI ADB, DTB     ; copy RW0 bytes src->dest
#endm

    SEGCOPY INIT, DCONST ; from DCONST to INIT
    SEGCOPY DIRINIT, DIRCONST ; from DIRCONST to DIRINIT
    SEGCOPY LIBINIT, LIBDCONST ; from LIBDCONST to LIBINIT
#if CONSTDATA == RAMCONST
    SEGCOPY CINIT, CONST ; from CONST to CINIT
#endif
#endif

```

Application Note

```
=====
; Clear uninitialized near data areas to zero
=====
#macro   SEGZERO SEC
        MOV  A,#BNKSEC \SEC      ; get bank of section
        MOV  ADB,A              ; store bank in ADB
        MOVW RW0,#SIZEOF (\SEC) ; store number of bytes in RW0
        MOVW A,\SEC             ; move dest offset to AL
        MOVN A,#0               ; move fill value to AL and
                                ; move AL (offset) to AH
        FILSI ADB               ; fill RW0 bytes with AL
#endm

        SEGZERO DATA          ; clear DATA
        SEGZERO DIRDATA        ; clear DIRDATA
        SEGZERO LIBDATA        ; clear LIBDATA

=====
; Copy initial value of far data areas.
; Each C-module has its own far INIT section. The names are generic.
; DCONST_module contains the initializers for the far data of the one
; module. INIT_module reserves the RAM area, which has to be loaded
; with the data from DCONST_module. ("module" is the name of the *.c
; file)
; All separated DCONST_module/INIT_module areas are described in
; DTRANS section by start addresses and length of each far section.
; 0000 1. source address (ROM)
; 0004 1. destination address (RAM)
; 0008 length of sections 1
; 000A 2. source address (ROM)
; 000E 2. destination address (RAM)
; 0012 length of sections 2
; 0014 3. source address ...
=====
        MOV  A, #BNKSEC DTRANS ; get bank of table
        MOV  DTB, A            ; store bank in DTB
        MOVW RW1, #DTRANS      ; get start offset of table
        OR   CCR, #H'20        ; System stack flag set (SSB used)
        BRA  LABEL2            ; branch to loop condition

LABEL1:
        MOVW A, @RW1+6         ; get bank of destination
        MOV  SSB, A            ; save dest bank in SSB
        MOVW A, @RW1+2         ; get source bank
        MOV  ADB, A            ; save source bank in ADB
        MOVW A, @RW1+4         ; move destination addr in AL
        MOVW A, @RW1           ; AL ->AH, src addr -> AL
        MOVW RW0, @RW1+8       ; number of bytes to copy -> RW0
        MOVSI SPB, ADB         ; copy data
        MOVN A, #10            ; length of one table entry is 10
        ADDW RW1, A            ; set pointer to next table entry

LABEL2:
        MOVW A, RW1            ; get address of next block
        SUBW A, #DTRANS        ; sub address of first block
        CMPW A, #SIZEOF (DTRANS) ; all blocks processed ?
        BNE  LABEL1            ; if not, branch

=====
; Clear uninitialized far data areas to zero
; Each C-module has its own far DATA section. The names are generic.
; DATA_module contains the reserved area (RAM) to be cleared.
; ("module" is the name of the *.c file)
; All separated DATA_module areas are described in DCLEAR section by
; start addresses and length of all far section.
; 0000 1. section address (RAM)
; 0004 length of section 1
; 0006 2. section address (RAM)
; 000A length of section 2
; 000C 3. section address (RAM)
; 0010 length of section 3 ...
=====
        MOV  A, #BNKSEC DCLEAR ; get bank of table
        MOV  DTB, A            ; store bank in DTB
        MOVW RW1, #DCLEAR      ; get start offset of table
        BRA  LABEL4            ; branch to loop condition

LABEL3:
        MOV  A, @RW1+2         ; get section bank
```

```

MOV  ADB, A           ; save section bank in ADB
MOVW RW0, @RW1+4     ; number of bytes to copy -> RW0
MOVW A, @RW1         ; move section addr in AL
MOVN A, #0           ; AL ->AH, init value -> AL
FILSI  ADB           ; write 0 to section
MOVN A, #6           ; length of one table entry is 6
ADDW RW1, A          ; set pointer to next table entry

LABEL4:
MOVW A, RW1          ; get address of next block
SUBW A, #DCLEAR      ; sub address of first block
CMPW A, #SIZEOF (DCLEAR) ; all blocks processed ?
BNE  LABEL3          ; if not, branch

;=====
; Prepare stacks and set the default stack type
;=====
#macro SYSSTACKINI
    OR  CCR, #H'20      ; set System stack flag
    MOV A, #BNKSYM SSTACK_TOP ; System stack set
    MOV SSB, A
    MOVW A, #SSTACK_TOP
    MOVW SP, A
#endm
#macro USRSTACKINI
    AND CCR, #H'DF      ; User stack flag set
    MOV A, #BNKSYM USTACK_TOP ; User stack set
    MOV USB, A
    MOVW A, #USTACK_TOP
    MOVW SP, A
#endm
#if STACKUSE == USRSTACK
    SYSSTACKINI
    USRSTACKINI          ; finally user stack selected
#else
    USRSTACKINI
    SYSSTACKINI          ; finally system stack selected
#endif

;=====
; Set default data bank and direct page register
;=====
    MOV  A, #BNKSEC DATA      ; User data bank offset
    MOV  DTB, A

    MOV  A, #PAGE DIRDATA_S    ; User direct page
    MOV  DPR, A

;=====
; Wait for PLL to stabilize
;=====

#if CLOCKSPEED > MAINCLOCK && CLOCKWAIT == ON
no_PLL_yet:
    BBS  I:CKSCR:6, no_PLL_yet ; check MCM and wait for
                                ; PLL to stabilize
#endif

;=====
; Call lib init function: reload stack afterwards, if AUTO-model
;=====
#if CLIBINI == DOLIBINIT
# if MEMMODEL == SMALL || MEMMODEL == COMPACT
    CALL __stream_init      ; initialize library IO
# else
    CALLP __stream_init     ; initialize library IO
# if MEMMODEL == AUTO
    RELOAD_SP               ; repair stack since stream_init was
                            ; possibly left by RET (not RETP)
# endif ; AUTO
# endif ; MEDIUM, LARGE, AUTO
#endif ; LIBINI

;=====
; Call C-language main entrance
;=====
#if MEMMODEL == SMALL || MEMMODEL == COMPACT

```

Application Note

```
        CALL _main          ; Start main function
#else
        CALLP _main        ; MEDIUM, LARGE, AUTO
                          ; Start main function
                          ; ignore remaining word on stack,
                          ; if main was completed by RET
#endif
#if CLIBINI == DOLIBINIT
# if MEMMODEL == SMALL || MEMMODEL == COMPACT
        CALL _exit
# else
        CALLP _exit       ; MEDIUM, LARGE, AUTO
                          ; ignore remaining word on stack,
                          ; if main was completed by RET
# endif
__exit:
#endif
end:
        BRA end           ; Loop

        .END notresetyet  ; define debugger start address
;=====
; End of startup
;=====
```

3.2 History

12 th May 99	Holger Lösche	Original version
17 th May 99	Holger Lösche	Minor typos corrected