Start mechanisms in ARM Cortex M-boards for C programs using GNU compiler collection.

(Example for a NXP LPC1113F301 board)

General aspects:

Before a user programs main() routine can start, there have some preparations to be done by the MCU hardware and by the preparing code which is provided by the C compiler and the Startup Code (here: **startup_LPC11xx.S**) and the SystemInit() routine in (here: **system_LPC11xx.c**) The final step is a BL.W jump to the main() routine of the user code.

What the MCU must do as 'native' functions:

Without any code loaded to a MCU board there are some basic functions that a board must provide:

1) The board must be flashable. That means that it must be able to receive data from an external device in a special manner and to write these data in its flash-memory (usually starting at address 0000.0000).
2) After the data have been loaded to the board the first 4 bytes hold the address where the Stackpointer shall initially point to (memory address in the RAM-area of the MCU), the next 4 bytes hold the address of the Reset_Handler in the Flash area of the MCU. When the board is reset, the board transfers the first address from location 0000.0000 into the MCU's Stackpointer, the second address from location 0000.0004 into the MCU's Program Counter. From there the further initialization of the board starts.

What Compiler and Linker must do to provide a valid image that can be flashed to the MCU:

Some Symbols must be defined which hold information needed by the initialization code. The code can access these symbols in the Startup Code (usually in a source file named like '**startup_lpc11xx.S**') through their names.

These symbols are defined through the gcc compiler itself, the linker script (here: **lpc1113F_301.ld**) and the Startup file (here: **startup_LPC11xx.S**)

The Startup Code needs the following Symbols:

_start	address of a codesegment called **_mainCRTStartup** lable in the flash memory
__etext	End of code section, i.e., begin of data sections to copy from
__data_start__	Start of RAM address range that data should be copied to
__data_end__	End of RAM address range that data should be copied to
__STARTUP_CLEAR_BSS	Defines if BSS Region (not initially used Ram) shall be cleared
__bss_start__	start of BSS section
__bss_end__	end of BSS section
__NO_SYSTEM_INIT	Defines if SystemInit() routine shall be called in Startup Code
__START	Should hold the address of label **_mainCRTStartup**. Is Initially not defined, so I defined it in the linker script and set it to 0000.0000. In the Startup Code then I set **__START** to _start.

The code following the '_mainCRTStartup' lable must know the startaddress of the user code main() routine.
(As I didn't find a place from where this address is taken I assume that the compiler writes this address directly into the produced code.)

An important symbol is **'_start**'. It holds the address of the Reset_Handler, from where the initialization process starts.

Steps in the initialization mechanism following a reset

1) The processor fetches the address of the **Reset_Handler** from addr. 0000.0004, transfers address into the Program Counter and executes the code of the Reset_Handler.
2) The Reset-Handler Code copies the content of the data section from the flash memory, where it was stored when the board was flashed, to its target location in the Ram of the MCU.
3) The Reset_Handler clears the not used Ram of the MCU.
4) The Reset_Handler calls the **SystemInit()** routine (in '**system_LPC11xx.c**'**)**, which makes some basic initialization like setting the clock rates).
5) The code of the Reset_Handler then jumps to the lable '**_mainCRTStartup**', does some more initializations and finally jumps to the start address of the users **main()** routine.

The linker script

The linker script (here: **lpc1113F_301.ld** in the folder Startup**)** instructs the gcc linker how to pack the different compiled components (C libraries, Startup script, system_LPC11xx.o and some needed symbols) together to an .elf file which then can be flashed to the board.
If someone wants to see the binary code which finally resides on the board, the utility 'fromelf' of the GNU compiler collection can be used.