

Building Bare-Metal ARM Systems with GNU

Part 2: Startup Code and the Low-level Initialization

Miro Samek

In this part I start digging into the code that is available online at [<provide embedded.com link to code>](#). The code contains C and C++ versions of the example application called “Blinky”, because it blinks the 4 user LEDs of the Atmel AT91SAM7S-EK evaluation board. The C version is located in the subdirectory `c_blinky`, and the equivalent C++ version is located in the subdirectory `cpp_blinky`. The Blinky application is primitive, but is carefully designed to use all the features covered in this multi-part article. The projects are based on the latest CodeSourcery G++ [GNU toolchain for ARM](#) [1].

In this part, I describe the generic startup code for the GNU toolchain as well as the low-level initialization for a bare-metal ARM system. The recommended reading for this part includes the “IAR Compiler Reference Guide” [2], specifically sections “System startup and termination” as well as “Customizing system initialization”.

1. The Startup Code

The startup sequence for a bare-metal ARM system is implemented in the assembly file `startup.s`, which is identical for C and C++ projects. This file is designed to be generic, and should work for any ARM-based MCU without modifications. All CPU- and board-specific low-level initialization that needs to occur before entering the `main()` function should be handled in the routine `low_level_init()`, which typically can be written in C/C++, but can also be coded in assembly, if necessary.

```
/*
 * The startup code must be linked at the start of ROM, which is NOT
 * necessarily address zero.
 */
(1) .text
(2) .code 32

(3) .global _start
(4) .func _start

_start:

    /* Vector table
     * NOTE: used only very briefly until RAM is remapped to address zero
     */
(5) B _reset          /* Reset: relative branch allows remap */
(6) B .              /* Undefined Instruction */
    B .              /* Software Interrupt */
    B .              /* Prefetch Abort */
    B .              /* Data Abort */
    B .              /* Reserved */
    B .              /* IRQ */
    B .              /* FIQ */

    /* The copyright notice embedded prominently at the beginning of the ROM */
(7) .string "Copyright (c) YOUR COMPANY. All Rights Reserved."
(8) .align 4         /* re-align to the word boundary */

/*
 * _reset
 */
```

```

*/
(9) _reset:

    /* Call the platform-specific low-level initialization routine
    *
    * NOTE: The ROM is typically NOT at its linked address before the remap,
    * so the branch to low_level_init() must be relative (position
    * independent code). The low_level_init() function must continue to
    * execute in ARM state. Also, the function low_level_init() cannot rely
    * on uninitialized data being cleared and cannot use any initialized
    * data, because the .bss and .data sections have not been initialized yet.
    */
(10) LDR    r0,=_reset      /* pass the reset address as the 1st argument */
(11) LDR    r1,=_cstartup  /* pass the return address as the 2nd argument */
(12) MOV    lr,r1          /* set the return address after the remap */
(13) LDR    sp,=__stack_end /* set the temporary stack pointer */
(14) B     low_level_init /* relative branch enables remap */

/* NOTE: after the return from low_level_init() the ROM is remapped
* to its linked address so the rest of the code executes at its linked
* address.
*/

(15) _cstartup:
    /* Relocate .fastcode section (copy from ROM to RAM) */
(16) LDR    r0,=__fastcode_load
    LDR    r1,=__fastcode_start
    LDR    r2,=__fastcode_end

1:
    CMP    r1,r2
    LDMLTIA r0!,{r3}
    STMLTIA r1!,{r3}
    BLT    1b

    /* Relocate the .data section (copy from ROM to RAM) */
(17) LDR    r0,=__data_load
    LDR    r1,=__data_start
    LDR    r2,=_edata

1:
    CMP    r1,r2
    LDMLTIA r0!,{r3}
    STMLTIA r1!,{r3}
    BLT    1b

    /* Clear the .bss section (zero init) */
(18) LDR    r1,=__bss_start__
    LDR    r2,=__bss_end__
    MOV    r3,#0

1:
    CMP    r1,r2
    STMLTIA r1!,{r3}
    BLT    1b

(19) /* Fill the .stack section */
    LDR    r1,=__stack_start__
    LDR    r2,=__stack_end__
    LDR    r3,=STACK_FILL

1:
    CMP    r2,r1
    STMLTIA r1!,{r3}
    BLT    1b

(20) /* Initialize stack pointers for all ARM modes */
    MSR    CPSR_c, #(IRQ_MODE | I_BIT | F_BIT)
    LDR    sp,=__irq_stack_top__ /* set the IRQ stack pointer */

    MSR    CPSR_c, #(FIQ_MODE | I_BIT | F_BIT)

```

```

LDR    sp,=__fiq_stack_top__          /* set the FIQ stack pointer */
MSR    CPSR_c,#(SVC_MODE | I_BIT | F_BIT)
LDR    sp,=__svc_stack_top__         /* set the SVC stack pointer */
MSR    CPSR_c,#(ABT_MODE | I_BIT | F_BIT)
LDR    sp,=__abt_stack_top__         /* set the ABT stack pointer */
MSR    CPSR_c,#(UND_MODE | I_BIT | F_BIT)
LDR    sp,=__und_stack_top__         /* set the UND stack pointer */
(21)   MSR    CPSR_c,#(SYS_MODE | I_BIT | F_BIT)
LDR    sp,=__c_stack_top__           /* set the C stack pointer */

/* Invoke all static constructors */
(22)   LDR    r12,=__libc_init_array
MOV    lr,pc                         /* set the return address */
BX     r12                             /* the target code can be ARM or THUMB */

/* Enter the C/C++ code */
(23)   LDR    r12,=main
MOV    lr,pc                         /* set the return address */
BX     r12                             /* the target code can be ARM or THUMB */

(24)   SWI    0xFFFFF                 /* cause exception if main() ever returns */

.size  _start, . - _start
.endfunc

.end

```

Listing 1 Startup code in GNU assembly (startup.s)

Listing 1 shows the complete startup code in assembly. The highlights of the startup sequence are as follows:

(1) The `.text` directive tells GNU assembler (`as`) to assemble the following statements onto the end of the text subsection.

(2) The `.code 32` directive selects the 32-bit ARM instruction set (the value 16 selects THUMB). The ARM core starts execution in the ARM state.

(3) The `.global` directive makes the symbol `_start` visible to the GNU linker (`ld`).

(4) The `.func` directive emits debugging information for the function `_start`. (The function definition must end with the directive `.endfunc`).

(5) Upon reset, the ARM core fetches the instruction at address `0x0`, which at boot time must be mapped to a non-volatile memory (ROM). However, later the ROM might be remapped to a different address range by means of a memory remap operation. Therefore the code in ROM is typically linked to the final ROM location and not to the ROM location at boot time. This dynamic changing of the memory map has at least two consequences. First, the few initial instructions must be position-independent meaning that only PC-relative addressing can be used. Second, the initial vector table is used only very briefly and is replaced with a different vector table established in RAM.

(6) The initial vector table contains just endless loops (relative branches to self). This vector table is used only very briefly until it is replaced by the vector table in RAM. Should an exception occur during this transient, the board is most likely damaged and the CPU cannot recover by itself. A safety-critical device should have a secondary circuit (such as an external watchdog timer driven by a separate clock source) that would announce the condition to the user.

(7) It is always a good idea to embed a prominent copyright message close to the beginning of the ROM image. You should customize this message for your company.

(8) Alignment to the word boundary is necessary after a string embedded directly in the code.

(9) The reset vector branches to this label.

(10) The `r0` and `r1` registers are used as the arguments of the upcoming call to the `low_level_init()` function. The register `r0` is loaded with the linked address of the reset handler, which might be useful to set up the RAM-based vector table inside the `low_level_init()` function.

(11) The `r1` register is loaded with the linked address of the C-initialization code, which also is the return address from the `low_level_init()` function. Some MCUs (such as AT91x40 with the EBI) might need this address to perform a direct jump after the memory remap operation.

(12) The link register is loaded with the return address. Please note that the return address is the `_cstartup` label at its final linked location, and not the subsequent PC value (so loading the return address with `LDR lr, pc` would be incorrect.)

(13) The temporary stack pointer is initialized to the end of the stack section. The GNU toolset uses the full descending stack meaning that the stack grows towards the lower memory addresses.

NOTE: The stack pointer initialized in this step might be not valid in case the RAM is not available at the linked address before the remap operation. It is not an issue in the AT91SAM7S family, because the RAM is always available at the linked address (0x00200000). However, in other devices (such as AT91x40) the RAM is not available at its final location before the EBI remap. In this latter case you might need to write the `low_level_init()` function in assembly to make sure that the stack pointer is not used until the memory remap.

(14) The function `low_level_init()` is invoked with a relative branch instruction. Please note that the branch-with-link (BL) instruction is specifically NOT used because the function might be called not from its linked address. Instead the return address has been loaded explicitly in the previous instruction.

NOTE: The function `low_level_init()` can be coded in C/C++ with the following restrictions. The function must execute in the ARM state and it must not rely on the initialization of `.data` section or clearing of the `.bss` section. Also, if the memory remapping is performed at all, it must occur inside the `low_level_init()` function because the code is no longer position-independent after this function returns.

(15) The `_cstartup` label marks the beginning of C-initialization.

(16) The section `.fastcode` is used for the code executed from RAM. Here this section is copied from ROM to its linked address in RAM (see also the linker script).

(17) The section `.data` is used for initialized variables. Here this section is copied from its load address in ROM to its linked address in RAM (see also the linker script).

(18) The section `.bss` is used for uninitialized variables, which the C standard requires to be set to zero. Here this section is cleared in RAM (see also the linker script).

(19) The section `.stack` is used for the stacks. Here this section is filled with the given pattern, which can help to determine the stack usage in the debugger.

(20) All banked stack pointers are initialized.

(21) The User/System stack pointer is initialized last. All subsequent code executes in the System mode.

(22) The library function `__libc_init_array` invokes all C++ static constructors (see also the linker script). This function is invoked with the BX instruction, which allows state change to THUMB. This function is harmless in C.

(23) The `main()` function is invoked with the BX instruction, which allows state change to THUMB.

(24) The `main()` function should never return in a bare-metal application because there is no operating system to return to. In case `main()` ever returns, the Software Interrupt exception is entered, in which the user can customize how to handle this problem.

2. Low-Level Initialization

The function `low_level_init()` performs the low-level initialization, which always strongly depends on the specific ARM MCU and the particular memory remap operation. As described in the previous section, the function `low_level_init()` can be coded in C or C++, but must be compiled to ARM and cannot rely on the initialization of the `.data` section, clearing of the `.bss` section, or on C++ static constructors being called.

```
(1) #include <stdint.h> /* C-99 standard exact-width integer types */
(2) void low_level_init(void (*reset_addr)(), void (*return_addr)()) {
(3)     extern uint8_t __ram_start;
(4)     static uint32_t const LDR_PC_PC = 0xE59FF000U;
(5)     static uint32_t const MAGIC = 0xDEADBEEFU;
        AT91PS_PMC pPMC;

        /* Set flash wait state FWS and FMCN */
(6)     AT91C_BASE_MC->MC_FMR = ((AT91C_MC_FMCN) & ((MCK + 500000)/1000000 << 16))
        | AT91C_MC_FWS_1FWS;
(7)     AT91C_BASE_WDTC->WDTC_WDMR = AT91C_WDTC_WDDIS; /* Disable the watchdog */

(8)     /* Enable the Main Oscillator */ . . .
        /* Set the PLL and Divider and wait for PLL stabilization */. . .
        /* Select Master Clock and CPU Clock select the PLL clock / 2 */. . .

        /* Setup the exception vectors in RAM.
        * NOTE: the exception vectors must be in RAM *before* the remap
        * in order to guarantee that the ARM core is provided with valid vectors
        * during the remap operation.
        */
        /* setup the primary vector table in RAM */
(9)     *(uint32_t volatile *)&__ram_start + 0x00) = (LDR_PC_PC | 0x18);
        *(uint32_t volatile *)&__ram_start + 0x04) = (LDR_PC_PC | 0x18);
        *(uint32_t volatile *)&__ram_start + 0x08) = (LDR_PC_PC | 0x18);
        *(uint32_t volatile *)&__ram_start + 0x0C) = (LDR_PC_PC | 0x18);
        *(uint32_t volatile *)&__ram_start + 0x10) = (LDR_PC_PC | 0x18);
(10)    *(uint32_t volatile *)&__ram_start + 0x14) = MAGIC;
        *(uint32_t volatile *)&__ram_start + 0x18) = (LDR_PC_PC | 0x18);
        *(uint32_t volatile *)&__ram_start + 0x1C) = (LDR_PC_PC | 0x18);

        /* setup the secondary vector table in RAM */
(11)    *(uint32_t volatile *)&__ram_start + 0x20) = (uint32_t)reset_addr;
        *(uint32_t volatile *)&__ram_start + 0x24) = 0x04U;
        *(uint32_t volatile *)&__ram_start + 0x28) = 0x08U;
        *(uint32_t volatile *)&__ram_start + 0x2C) = 0x0CU;
        *(uint32_t volatile *)&__ram_start + 0x30) = 0x10U;
        *(uint32_t volatile *)&__ram_start + 0x34) = 0x14U;
        *(uint32_t volatile *)&__ram_start + 0x38) = 0x18U;
        *(uint32_t volatile *)&__ram_start + 0x3C) = 0x1CU;

        /* check if the Memory Controller has been remapped already */
(12)    if (MAGIC != *(uint32_t volatile *)&0x14) {
(13)        AT91C_BASE_MC->MC_RCR = 1; /* perform Memory Controller remapping */
    }
(14) }
```

Listing 2 Low-level initialization for AT91SAM7S microcontroller.

Listing 2 shows the low-level initialization of the AT91SAM7S microcontroller in C. Note that the initialization for a different microcontroller, such as AT91x40 series with the EBI, could be different mostly due to different memory remap operation. The highlights of the low-level initialization are as follows:

(1) The GNU gcc is a standard-compliant compiler that supports the C-99 standard exact-width integer types. The use of these types is recommended.

(2) The arguments of `low_level_init()` are as follows: `reset_addr` is the linked address of the reset handler and `return_addr` is the linked return address from the `low_level_init()` function.

NOTE: In the C++ environment, the function `low_level_init()` must be defined with the extern "C" linkage specification because it is called from assembly.

(3) The symbol `__ram_start` denotes the linked address of RAM. In AT91SAM7S the RAM is always available at this address, so the symbol `__ram_start` denotes also the RAM location **before** the re-map operation (see the linker script).

(4) The constant `LDR_PC_PC` contains the opcode of the ARM instruction `LDR pc, [pc, ...]`, which is used to populate the RAM vector table.

(5) This constant `MAGIC` is used to test if the remap operation has been performed already.

(6) The number of flash wait states is reduced from the default value set at reset to speed up the boot process.

(7) The AT91 watchdog timer is disabled so that it does not expire during the boot process. The application can choose to enable the watchdog after the `main()` function is called.

(8) The CPU and peripheral clocks are configured. This speeds up the rest of the boot process.

(9) The ARM vector table is established in RAM **before** the memory remap operation, so that the ARM core is provided with valid vectors at all times. The vector table has the following structure:

```
0x00:    LDR pc, [pc, #0x18]    /* Reset          */
0x04:    LDR pc, [pc, #0x18]    /* Undefined Instruction */
0x08:    LDR pc, [pc, #0x18]    /* Software Interrupt */
0x0C:    LDR pc, [pc, #0x18]    /* Prefetch Abort   */
0x10:    LDR pc, [pc, #0x18]    /* Data Abort       */
0x14:    LDR pc, [pc, #0x18]    /* Reserved         */
0x18:    LDR pc, [pc, #0x18]    /* IRQ vector       */
0x1C:    LDR pc, [pc, #0x18]    /* FIQ vector       */
```

All entries in the RAM vector table load the PC with the address located in the secondary jump table that immediately follows the primary vector table in memory. For example, the Reset exception at address 0x00 loads the PC with the word located at the effective address: 0x00 (+8 for pipeline) + 0x18 = 0x20, which is the address immediately following the ARM vector table.

NOTE: Some ARM MCUs, such as the NXP LPC family, remap only a small portion of RAM down to address zero. However, the amount of RAM remapped is always at least 0x40 bytes (exactly 0x40 bytes in case of LPC), which is big enough to hold both the primary vector table and the secondary jump table.

(10) The jump table entry for the unused exception is initialized with the `MAGIC` number. Please note that this number is written to RAM at its location **before** the memory remap operation.

(11) The secondary jump table in RAM is initialized to contain jump to `reset_addr` at 0x20 and endless loops for the remaining exceptions. For example, the Prefetch Abort exception at address 0x0C will cause loading the PC again with 0x0C, so the CPU will be tied up in a loop. This is just the temporary setting until the application initializes the secondary jump table with the addresses of the application-specific exception handlers. Until this happens, the application is not ready to handle the interrupts or exceptions, anyway.

NOTE: Using the secondary jump table has many benefits. First, the application can very easily change the exception handler by simply writing the handler's address in the secondary table, rather than synthesize a relative branch instruction at the primary vector table. Second, the load to PC

instruction allows utilizing the full 32-bit address space for placement of the exception handlers, whereas the relative branch instruction is limited to +/- 25 bits relative to the current PC.

(12) The word at the absolute address 0x14 is loaded and compared to the MAGIC number. The location 0x14 is in ROM before the remap operation, and is in RAM after the remap operation. Before the remap operation the location 0x14 contains the B instruction, which is different from the MAGIC value.

(13) If the location 0x14 does not contain the MAGIC value, this indicates that the write to RAM did not change the value at address 0x14. This, in turn, means that RAM has not been remapped to address 0x00 yet (i.e., ROM is still mapped to the address 0x00). In this case the remap operation must be performed.

NOTE: The AT91SAM7 Memory Controller remap operation is a toggle and it is impossible to detect whether the remap has been performed by examining any of the Memory Controller registers. The technique of writing to the low RAM address can be used to reliably detect whether the remap operation has been performed to avoid undoing it. This safeguard is very useful when the reset is performed during debugging. The soft-reset performed by a debugger typically does not undo the memory remap operation, so the remap should not be performed in this case.

(14) The `low_level_init()` function returns to the address set by the startup code in the `lr` register. Please note that at this point the code starts executing at its linked address.

3. Coming Up Next

In the next part I'll describe the linker script for the GNU toolchain. Stay tuned.

4. References

[1] GNU Assembler (as) HTML documentation included in the CodeSourcery Toolchain for ARM, http://www.codesourcery.com/gnu_toolchains/arm.

[2] IAR Systems, "ARM® IAR C/C++ Compiler Reference Guide for Advanced RISC Machines Ltd's ARM Cores", Part number: CARM-13, Thirteenth edition: June 2006. Included in the free EWARM KickStart edition <http://supp.iar.com/Download/SW/?item=EWARM-KS32>.

[3] Lewin A.R.W. Edwards, "Embedded System Design on a Shoestring", Elsevier 2003.

[4] ARM Projects, http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects

Miro Samek is the President of [Quantum Leaps, LLC](http://www.quantum-leaps.com), the provider of lightweight, state machine-based, event-driven frameworks for embedded systems. He is the author of "Practical Statecharts in C/C++" (CMP Books, 2002), has written numerous articles for magazines, and is a regular speaker at the Embedded Systems Conferences. He welcomes contact at miro@quantum-leaps.com.