

Introduction

This Application Note is aimed to describe the basic steps of the device boot flow of the SPC58xCx and SPC58xG8x devices, in order to support mostly software developers and application programmers in the design of applications to implement all the needed steps for using the microcontroller. It tries to be as much as possible independent from any adopted toolchain, thus without relying upon any runtime libraries for the start-up phase.

The boot flow here considered starts from the power-on and ends when the MCU has started execution of the user application. Considering the scope and goals of this document, the most important blocks of the MCU are the CPUs and the IPs involved on the reset process, that are shortly described later on. Further a brief description of the reset process is given.

To help with the understanding of the boot flow and the corresponding device configuration, some assembly and C code is included and commented purely as reference. It is based on a simple bare code distributed with the ST SPC5-STUDIO free tool available from ST website.

Despite SPC58xCx and SPC58xG8x are targeting different automotive application domains, and have some differences in terms of types/configurations and/or instances of peripheral IPs, different flash and system memory sizes, or targeting different safety levels (ASIL-B vs ASIL-D), or different packaging solutions, as far as the system start-up flow is concerned, they have many things in common so that can be described in a unique document. In case of specific settings are required depending on device characteristic then they are specifically highlighted.

Limitations

The initialization flow described in this application note doesn't include any safety/security topics as well as any special memory handling like internal memory remapping or memory protection levels.

It is intended that this application note is read along with the SPC58xCx/SPC58xG8x Reference Manuals, that can be obtained from the STMicroelectronics website at <http://www.st.com> (see [A.1](#): 2, 3).

Contents

1	Architecture overview	5
1.1	SPC58xCx architecture overview	5
1.2	SPC58xG8x architecture overview	6
1.3	MCU Cores features	8
1.4	Reset and boot	9
2	SPC58xCx/SPC58xG8x system start-up	13
2.1	Start-up flow (high-level view)	14
2.2	Start-up code explained	15
2.2.1	System initialization	15
2.2.2	EABI “C runtime” initialization	21
2.2.3	Early Platform initialization	22
2.2.4	DATA sections handling	30
2.2.5	Late platform initialization and main jump	31
2.3	Multicore system startup	32
2.3.1	Multi-core boot (by CPU)	32
2.3.2	Multi-core boot (by BAF)	34
Appendix A	Further Information	35
A.1	Reference documents	35
A.2	Acronyms and abbreviations	35
	Revision history	36

List of tables

Table 1.	SPC58ECx core memory configuration	9
Table 2.	SPC58NG8x core memory configuration	9
Table 3.	Boot record structure searched by the SSCM	10
Table 4.	Boot record structure searched by the BAF	11
Table 5.	CPU enable flags	12
Table 6.	Chip modes of operation.	27
Table 7.	ME_CADDRx/ME_CCTLx registers mapping.	33
Table 8.	Acronyms and abbreviations	35
Table 9.	Document revision history	36

List of figures

Figure 1. SPC58ECx Block diagram 6

Figure 2. SPC58NG8x Block diagram 8

Figure 3. Boot CPU start-up sequence 11



1 Architecture overview

In this chapter it is described a short overview of the two devices firstly. Then some more detailed information on the core configuration is provided as they are key for the goal of this application note.

1.1 SPC58xCx architecture overview

The SPC58xCx device is a 32-bit Power Architecture microcontroller for automotive ASIL-B applications, targeting vehicle body and gateway such as:

- Central body controller
- Smart junction box
- Mid and high end gateway

The device in the full featured configuration is equipped with two symmetrical e200z420 CPUs @180 MHz (named core_0 and core_2), featured with Local D-RAM (D-MEM), Instruction (I-Cache) and Data Cache (D-Cache) to increase system performance. It also includes enhancements aimed to fulfil safety requirement (e2e ECC, core MPU).

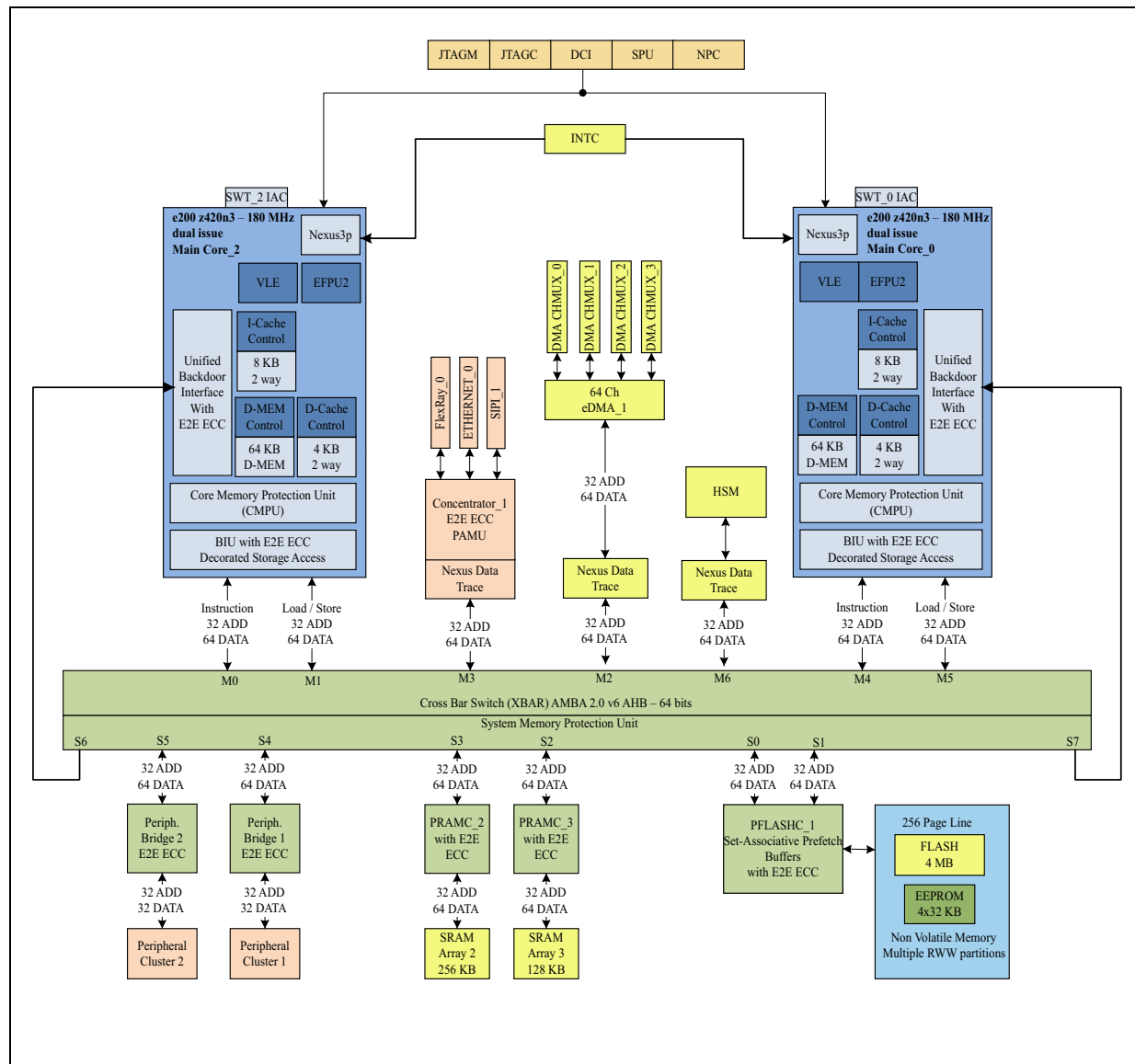
It also includes one e200z0 CPU @100 MHz into the Hardware Security Module (HSM).

Apart of the processor local memory and caches, the platform is equipped with on-chip flash memory (and related flash controller) and on-chip general –purpose System SRAM.

Both CPUs are connected to the external memories and I/O peripheral devices through a 64-bit Cross Bar Switch (XBAR). For a detailed description of all the devices and interfaces and all the other features implemented by the platform refer to the SPC58xCx Reference Manual (see [A.1](#): 1, 2).

The top-level block diagram below gives an overview of the microcontroller architecture.

Figure 1. SPC58ECx Block diagram



1.2 SPC58xG8x architecture overview

The SPC58xG8x device is a 32-bit Power Architecture microcontroller for automotive ASIL-D applications, targeting body automotive domain, more specifically:

- Integrated chassis control, high-end steering and braking
- High-end central body controller, smart junction boxes, stand-alone and integrated gateway
- In general any safety critical application requiring a very high level of safety integrity

Differently from the SPC58xCx device, the SPC58xG8x in the full featured configuration is equipped with six PowerPC CPUs (some in Delayed Lock-Step mode) distributed across two different computing shells and the HSM as below:

- Computing Shell 0
 - 2 x e200z420 @180 MHz (namely core_0 and core_1)
 - 1 x e200z419 @180 MHz (named core_0 checker) paired to core_0 in delayed lock-step mode
- Computing Shell 1
 - 1 x e200z425 @180 MHz (named core_2) with enhancements for DSP algorithms
 - 1 x e200z425 @180 MHz (named core_2 checker) paired to core_2 in delayed lock-step mode
- Hardware Security Module (HSM)
 - 1 x e200z0 @90 MHz

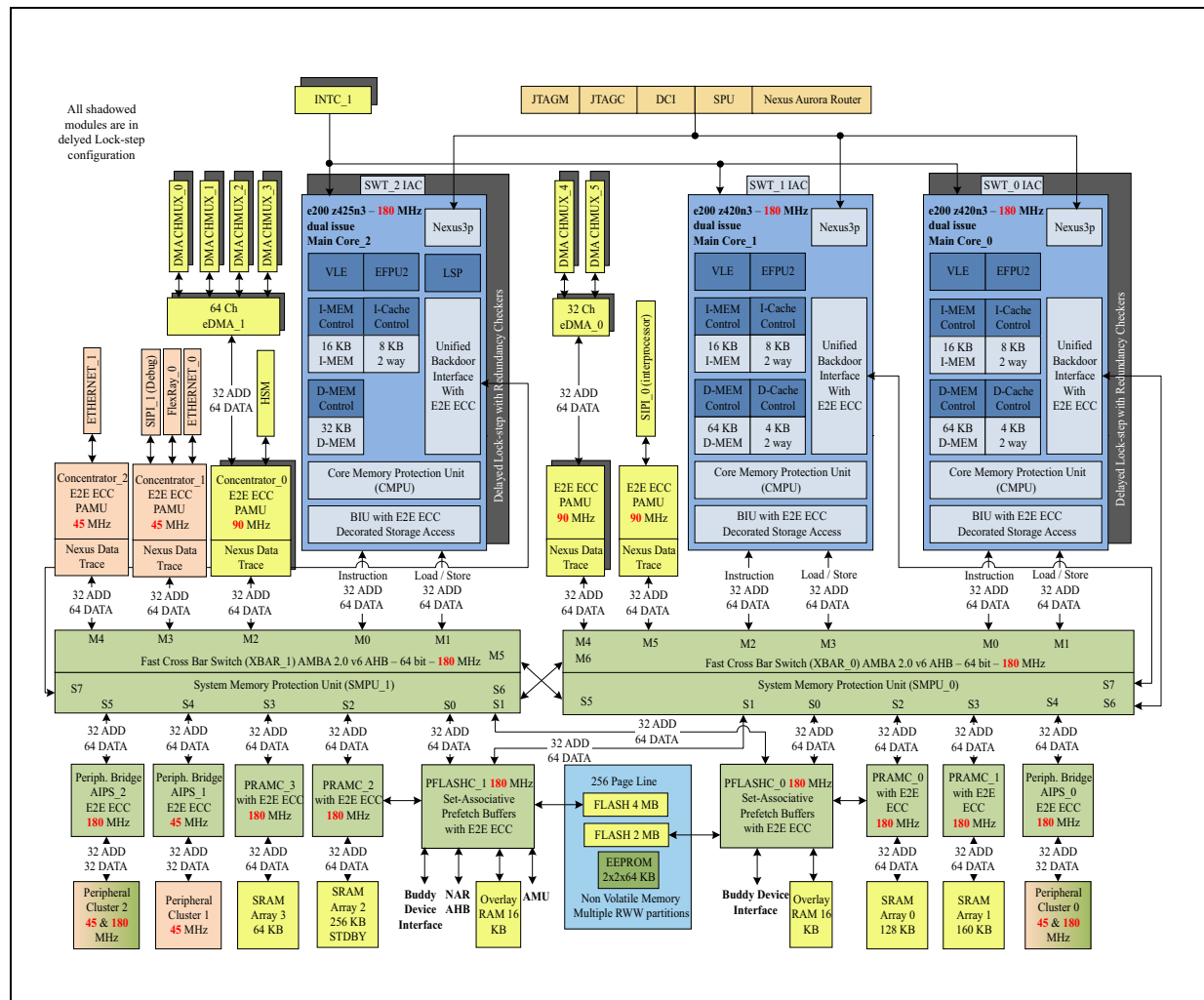
Differently from the cores available into the SPC58xCx device, the configuration of the six cores on SPC58xG8x MCU differs in terms of local memory and cache configuration.

This has some impacts on the configuration to be implemented by the software boot code as it is highlighted later.

Both Computing Shell 0 and 1 are connected to all the peripherals through two identical Fast Cross Bar Switches, 64-bit wide @180MHz, named respectively XBAR_0 and XBAR_1. A detailed description of the whole MCU architecture is out of the scope of such application note; for a fully comprehensive view refer to the related Reference Manual (see [A.1: 2, 3](#)).

The top-level block diagram below gives an overview of the microcontroller architecture.

Figure 2. SPC58NG8x Block diagram



1.3 MCU Cores features

All CPUs are compatible with the Power Architecture VLE instruction set which supports some code size reduction. The Power Architecture has enhancements that improve the architecture's fit in embedded applications.

From a software perspective, every bus master device (CPUs, DMA, SIPI, Ethernet, FlexRay) sees each memory (RAM modules, Flash) and peripherals in a consistent, flat memory map.

As reported in previous sections, the cores available in the computing shells on both MCUs are not perfectly identical. They have some slightly different configurations that have impacts on the configuration to be implemented into the software start-up code, that is the main focus of this application note, so it is worth to highlight what the differences are in terms on CPU local memory (D-MEM and/or I-MEM) and CPU cache (D-Cache and/or I-Cache). In the tables below it is possible to have an overall view of the main differences across all the CPU instances.

Table 1. SPC58ECx core memory configuration

Core instance	Core type	D-MEM	I-MEM	D-Cache	I-Cache
core_0, core_2	e200z420n3	64 KB	None	4 KB	8 KB
HSM core	e200z0	None	None	None	none

Table 2. SPC58NG8x core memory configuration

Core instance	Core type	D-MEM	I-MEM	D-Cache	I-Cache
core_0, core_1	e200z420n3	64 KB	16 KB	4 KB	8 KB
core_2	e200z425n3	32 KB	16 KB	None	8 KB
Checker core_0	e200z419	None	None	None	None
Checker core_2	e200z425n3	32 KB	16 KB	None	8 KB
HSM core	e200z0	None	None	None	None

As it can be seen from the [Table 2](#), on SPC58xG8x device the boot code running on core_0 and core_1 needs to take into account the presence of the D-Cache that is not available on the core_2. And also in order to re-use the same start-up software to boot-up core_0 and core_2 on the SPC58xCx micro, it needs to take into account that the I-MEM is not present (see [Table 1](#)). This different memory configuration has impacts on the software memory map as the start-up code cannot be loaded into the I-MEM to gain better performance.

1.4 Reset and boot

The power-up reset sequence always begins with the application of the power and follows different sequences depending on the condition of the SPC58xCx/SPC58xG8x family devices and various chip modes enabled from settings in the DCF records.

There are several modules used to step through the sequence needed to properly reset the SPC58xCx/SPC58xG8x and prepare it to fetch the first instruction of the user application code. These are briefly described here thereafter:

- Power Management Controller (PCM): it controls and monitors various voltage levels around the chip
- Reset Generation Module (RGM): it's a state machine, not executing program code. It centralizes the various reset sources and manages the reset sequence.
- System Status and Configuration Module (SSCM): it's a state machine as well. It's in charge to deliver the initial device configuration. It has a role in the boot-up process as responsible to locate the boot vector from flash device (for boot core and eventually for the HSM secure core)
- Mode Entry Module (MC_ME): it configures device execution mode and delivers reset vectors to all cores. It has an important role in the multi-core boot-up configuration.
- Self-Test Control Unit (STCU2): it runs the Logic Built-in Self-Test (LBIST) and Memory Built-in Self-Test (MBIST) for safety needs.

The reset procedure is a complex flow that involves all module above and each of them evolves through the respective finite machine states. Each module depending on the stage of the reset could provide different trigger signals to the other module, so that starting from

the power-on, the MCU will reach into the IDLE[FUNC] state where finally the MCU is able to executing code.

For a more detailed description of the reset flow refer to the chapter 8 of the Reference Manual (see [A.1](#): 2, 3).

In fact at the end of the reset sequence, there is always only one core (named Boot CPU) that is automatically woken-up (optionally the secure HSM core is started as well, depending on the hardware configuration). On Chorus devices the role of Boot CPU is played by Core_2. It then performs all the needed system initialization tasks and once done it is ready to execute user application.

In the next section the system boot-up code will be described assuming a single core boot configuration. Refer to the dedicated section for the multicore use case.

The Core_2 reset vector, read by the SSCM from the DCF and then transferred to the core by mean of the MC_ME module, will always point to a location into the Boot Assist Flash. Depending on the system configuration stored into the DCF BAF record into the flash, the Boot CPU will either execute the BAF code (located in a specific location into a flash block) or will start executing the user application code directly. In the latter case, the application boot address is specified into the so called SSCM boot header, whose structure is described below:

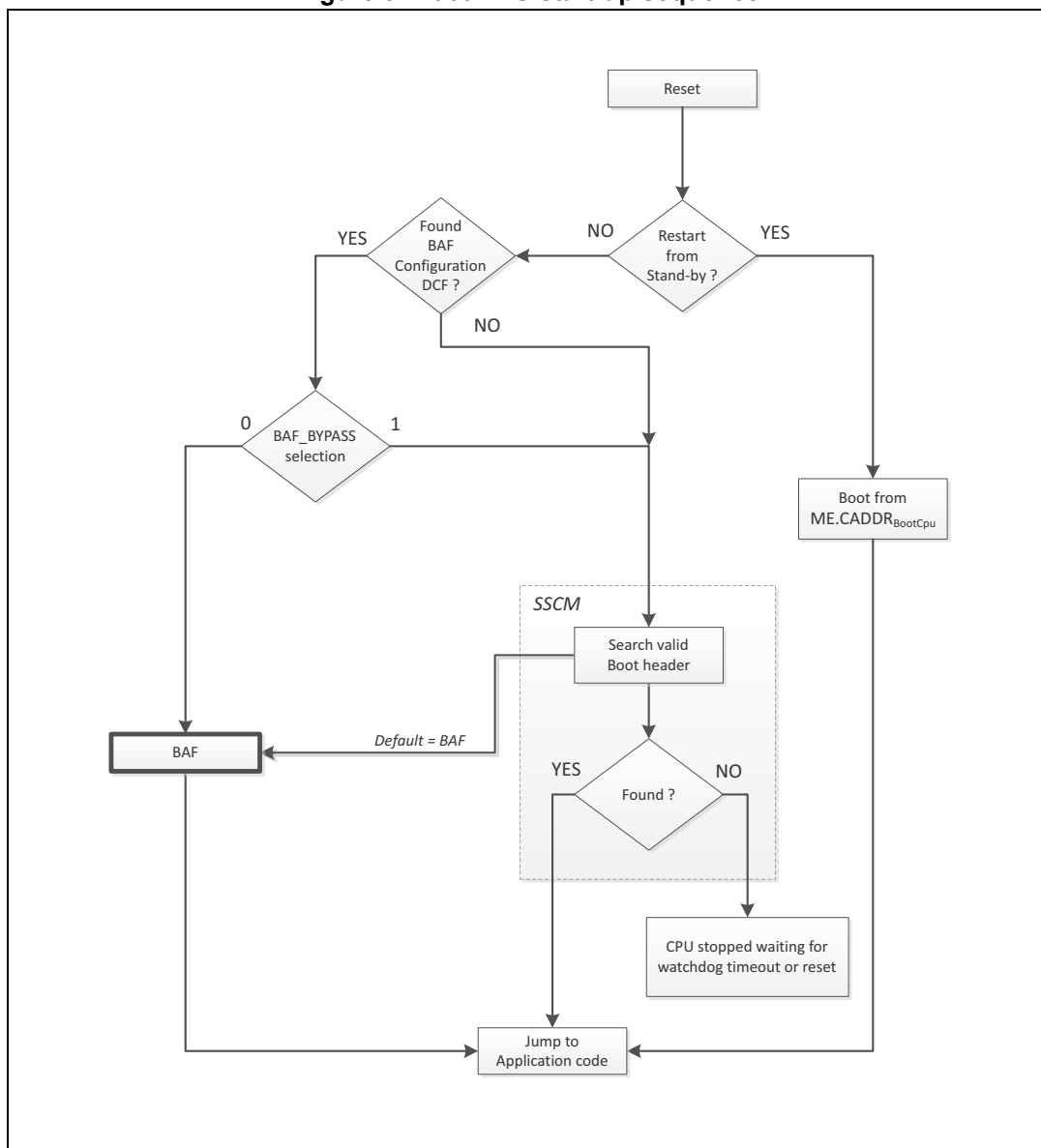
Table 3. Boot record structure searched by the SSCM

Offset	Field description	
0x4	Application start address	
0x0	SSCM Boot Record Tag (A5h)	Reserved

In case of boot w/o BAF support, the only CPU that is able to execute the application code is the Boot CPU. Application code will be in charge to execute all the steps required to wake up the other secondary cores in case of a multicore application. This will be described later.

If the BAF code is instead executed, the boot address is retrieved by the software in a similar way done by the SSCM. For detail on the lookup algorithm refer to the corresponding Reference Manual (see [A.1](#): 2, 3).

Figure 3. Boot CPU start-up sequence



The BAF Boot header has a slightly different structure compared to the SSMC boot record as shown in [Figure 3](#). In this case it is possible to have a more flexible boot configuration. It is possible to specify which CPU will be woken-up by writing the CPU enable flags accordingly and define different boot address for the different CPUs.

The BAF Boot Record structure is defined below.

Table 4. Boot record structure searched by the BAF

Offset	Field description
0xC	CPU_1 Reset vector
0x8	CPU_0 Reset vector

Table 4. Boot record structure searched by the BAF (continued)

Offset	Field description	
0x4	Boot CPU (CPU_2) Reset vector	
0x0	BAF Boot Record Tag (A5h)	CPU enable flags (see Table 5)

Table 5. CPU enable flags

16-19	20-23	24-27	28	29	30	31
0000	0000	0000	0	CPU_1	CPU_0	CPU_2

2 SPC58xCx/SPC58xG8x system start-up

In this section it will be described all the basic and most important initializations that are required to setup the microcontrollers for executing end-user application. In order to focus only on the platform initialization, in this section it will be considered the single core boot scenario only.

As previously mentioned, the CPU reset vectors are retrieved by hardware (via SSCM) or by software (via BAF code) by inspecting the BAF record. So the code image burned into the flash partition needs to provide a proper boot record as described below:

SPC58xG8x Boot record

```
.section .boot, "axv"

.long    0x00A50001    /* Only Core 2 will run
                        0x00A50007: Core 0,1 woken-up too
                        0x00A50005: Core 1 woken-up too
                        0x00A50003: Core 0 woken-up too
                        */

.long    _reset_address2 /* Core 2 reset address. */
.long    _reset_address0 /* Core 0 reset address. */
.long    _reset_address1 /* Core 1 reset address. */
```

SPC58xCx Boot record

```
.section .boot, "axv"

.long    0x00A50001    /* Only Core 2 will run
                        0x00A50003: Core 0 woken-up too
                        */

.long    _reset_address2 /* Core 2 reset address. */
.long    _reset_address0 /* Core 0 reset address. */
```

In the single-core boot use case, the code will look like the code above. The reset address for the other cores is assumed to be available but actually the tag is for one core only.

By providing the proper tag with all the CPU enable flags set, the BAF code will be able to wake-up all the CPUs and manage the delivery of the start-up address.

The start-up flow is described here thereafter at high level. Then a detailed explanation is provided for each steps. Further in order to clarify better all the steps needed to initialize and configure the device, a reference code is also attached implementing the start-up process.

2.1 Start-up flow (high-level view)

The start-up code is split into five main parts

- **System initialization:** it concerns all the initialization strictly related to the core itself (local memories, caches, exception, ...) and system memory. These are the first instructions executed by the CPU once it has been enabled and the reset vector has been delivered to it. It is implemented in assembly code, into the boot.s file.
- **EABI “C runtime” initialization:** in order to allow a proper execution of application code in C language compiled with EABI compliant toolchain, some core registers must be properly initialized. This is done into the C runtime support assembly file crt.s that is the trampoline to the application entry point.
- **Early platform initialization:** this part concerns with the initialization of the microcontroller platform, covering mostly clocks, power mode, cross-bar, flash controller in order to guarantee that the microcontroller is properly configured. The initialization of all other needed peripherals is out of the scope of this application note and not part anyway of the start-up phase. The early initialization in our reference code is implemented in C source into the file clock.c
- **DATA section handling:** Once the basic initialization is done, last few things are needed before user application can be run. BSS ELF section must be cleared and initialized DATA sections must be copied from the flash device to the System SRAM. This is done again in assembly code into the crt.s trampoline. Code is structured to provide an hook to user application for implementing eventually and optionally any late initialization. After that the core will jump to the standard ‘main’ C function.
- **Late platform initialization:** This an optional step provided by the software infrastructure to perform any option system initialization, optimization and platform customization that could be aimed to support the specific end user application. Reference code distributed with this AN will just provide an empty implementation.

Below a more detailed description of the tasks executed into each of the five parts. In the next section it will be provided a more detailed view and the corresponding code extracted from the reference example.

- System initialization
 - Set PIR register
 - Clearing of all core registers
 - Clearing of System SRAM
 - Clearing of core D-MEM (and I-MEM for SPC58xG8x)
 - Branch Target Buffer initialization
 - Caches initialization
 - Exception vector initialization (IVPR and MSR)
 - Relocating image from FLASH into RAM (optional)
 - Setup of platform initialization code address
- EABI “C runtime” setup
 - Setup process stack
 - Initialize SDATA and SDATA2 registers
- Early Platform initialization
 - Disable SWT device (optional step)
 - SSCM Error handling configuration

- Reset sources clearing
- Clock configuration
 - Oscillator by-pass (optional step)
 - Clock tree initialization
 - Enabling XOSC
 - PLL configuration
- Chip mode of operations setup
- DATA section handling
 - Clearing BSS sections
 - Copying initialized DATA section from flash to RAM
- Late platform initialization

2.2 Start-up code explained

2.2.1 System initialization

2.2.1.1 Set PIR register

PIR (Processor ID Register) is the Special Purpose Register (SPR) 286. It needs to hold the CPU number identifier. It can be done by the following code:

```
e_li      %r0, n          /* n {0,1,2} is the CPU number */
mtspr    286, %r0        /* PIR register.*/
```

2.2.1.2 Clearing of all core registers

The reset values of the General Purpose Registers (GPRs) are undefined. In order to guarantee a clean startup of the application, it's a good practice to clear all the registers before any use. Later, according to the application EABI, some registers need to be initialized at some specific values. Clearing GPRs is a key step for the checker cores that are in lock-step mode. As they will replicate the operations performed by the replicated core, it's mandatory that the registers are set in a clear condition.

The assembly code for clearing GPRs, might look like this:

```
/* Clears the internal registers of the calling core. */
_regclear:
    xor     %r0, %r0, %r0
    xor     %r1, %r1, %r1

/* [SNIP] Repeat for r2 to r29*/

    xor     %r30, %r30, %r30
    xor     %r31, %r31, %r31
```

It's also useful to cleanup CPU registers that might be stacked. This is necessary also playing with lockstep core(s), when available.

```
mtspr 1,r1
mtcrf 0xFF,r1
mtspr CTR,r1
mtspr SPRG0,r1
mtspr SPRG1,r1
mtspr SPRG2,r1
mtspr SPRG3,r1
mtspr SRR0,r1
mtspr SRR1,r1
mtspr CSRR0,r1
mtspr CSRR1,r1
mtspr MCSRR0,r1
mtspr MCSRR1,r1
mtspr DEAR,r1
mtspr USPRG0,r1
mtspr 62,r1
```

2.2.1.3 Clearing of System SRAM

As the System SRAM is coupled with an ECC engine, before any access to memory locations, start-up cope must guarantee that the memory locations are cleared to avoid generating parity check errors due to garbage values. The assembly code implementing this step might look like:

```

/* System RAM clearing */

        .align      2

_sysraminit:

        /* Shared RAM.*/

        e_lis        %r4, HI(__ram_start__)
        e_or2i        %r4, LO(__ram_start__)

        e_lis        %r5, HI(__ram_end__)
        e_or2i        %r5, LO(__ram_end__)

.sramclrloop:
 NOVLE

        cml          cr0, %r4, %r5

 VLE

        se_bge        .sramclrend
        e_stmw        %r16, 0(%r4)
        e_addi        %r4, %r4, 64
        se_b          .sramclrloop

.sramclrend:

```

The code consists into a tight loop, where the memory content is cleared at block of 64 bytes, writing the content of (zero-ed) GPRs r16 to r31 using the multiple work store instruction. It clearly assumes that registers (r16 – r31) have been previously cleared as well.

The start and end SRAM addresses are represented by the linker variables ‘__ram_start’ and ‘__ram_end’ whose values are defined into the application linker script.

2.2.1.4 Clearing of core D-MEM (and I-MEM for SPC58xG8x)

Similarly to System SRAM, the local core memory (including D-MEM and I-MEM where applicable) needs to be cleared before attempting any access to its location. The start-up code will perform this step in a similar way to the SRAM, with the following piece of assembly code:

```

/* Core 2 RAM clearing */
    .align      2
_ram2init:
    /* Core 2 DRAM.*/
    e_lis       %r4, HI(__dram2_start__)
    e_or2i      %r4, LO(__dram2_start__)
    e_lis       %r5, HI(__dram2_end__)
    e_or2i      %r5, LO(__dram2_end__)
.dram2clrloop:
    _NOVLE
        cmpl     cr0, %r4, %r5
    _VLE
        se_bge   .dram2clrend
        e_stmw   %r16, 0(%r4)
        e_addi   %r4, %r4, 64
        se_b     .dram2clrloop
.dram2clrend:

```

Code implementation is pretty similar. The extracted code is applicable to any of the cores. The example is referring to the core_2 case, as it can be retrieved from the name of the linker variables defining the memory boundaries ('__dram2_start' and '__dram2_end'). Those symbols are defined by the corresponding linker script. A counterpart for Core_0 (and Core_1 in case of SPC58xG8x) must be implemented.

2.2.1.5 Branch Target Buffer initialization

All e200z420n3, e200z419, and e200z425n3 processor cores features a Branch Processing Unit used to predict and prefetch branch target address. In order to use it, start-up code must flush, invalidate, configure and enable it by the BTB control register; more specifically by writing into the BUCSR (Branch Unit Control and Status Register (SPR 1013)). It's possible to accomplish all the initialization by a single write into the register. Example code might look like:

```

/* BUCSR registers definitions */
#define BUCSR_BPEN                0x00000001
#define BUCSR_BALLOC_BBFI        0x00000200
/* BUCSR default settings */
#define BUCSR_DEFAULT             (BUCSR_BPEN | BUCSR_BALLOC_BBFI)
e_li      %r3, BUCSR_DEFAULT
mtspr     1013, %r3 /* BUCSR */

```

BPEN (BUCSR[31]) bit if set will enable BTB to predict branches.

BALLOC_BFI mask is used to set BBFI (BUCSR[22]) bit to Flush & Invalidate the BTB. It is also used to clear the BALLOC (BUCSR[26:27]) bits to allocate BTB for all branches (backward and forward).

2.2.1.6 Caches initialization

All main cores are equipped with 8 KB of instruction cache. So in order to speed-up program execution it's possible to use it since the early stage of the boot process. I-Cache must be invalidated firstly and then enabled. L1 I-Cache is controlled by L1CSR1 register (L1 Cache Control and Status Register 1), that is SPR 1011. ICE (L1CSR1[31]) bit if set will enable I-Cache. ICINV (L1CSR1[30]) bit if set to 1 will initiate an invalidation process in hardware. When the operation is completed the ICINV bit is reset to 0. It's also needed to enable the ECC capability by setting bit ICECE (L1CSR1[15]). Code will look like this.

```
#define L1CSR1_ICE                0x00000001
#define L1CSR1_ICINV              0x00000002
#define L1CSR1_ICECE              0x00020000
#define L1CSR1_DEFAULT            (L1CSR1_ICE | L1CSR1_ICECE)

/* Cache initialization */
_cacheinit:
    /* ICache invalidated and then enabled.*/
    e_li        %r3, L1CSR1_ICINV
    mtspr       1011, %r3        /* L1CSR1 */
.iclrwait:
    mfspr       %r3, 1011        /* L1CSR1 */
    e_and2i.    %r3, L1CSR1_ICINV
    se_bne      .iclrwait        /* Wait for ICINV bit to be cleared */
    e_lis       %r3, HI(L1CSR1_DEFAULT)
    e_or2i      %r3, LO(L1CSR1_DEFAULT)
    mtspr       1011, %r3        /* L1CSR1 */
```

2.2.1.7 Exception vector initialization (IVPR and MSR)

Another mandatory step in the start-up process is the initialization of the Interrupt Vector Prefix Register (IVPR) that is used to compute the starting address of the software interrupt handler, together with the IVORx (Interrupt Vector Offset Register) register.

The symbol ‘__ivpr2_base__’ is defined into the linker script. Similarly, the same initialization is done for all the other cores.

Another initialization is related to the Machine State Register (MSR). A suggested default initialization comprises the following features:

- Bit SPE (MSR[6]): if set, the SPE APU vector instructions execution is enabled
- Bit WE (MSR[13]): if set, Wait State (power management) is enabled
- Bit CE (MSR[14]): if set, Critical Interrupts and Watchdog timers interrupts are enabled
- Bit ME (MSR[19]): if set, Asynchronous Machine Check interrupts are enabled

The code to implement this initialization might look like this:

```
/* MSR register definitions */

#define MSR_SPE                0x02000000
#define MSR_WE                 0x00040000
#define MSR_CE                 0x00020000
#define MSR_ME                 0x00001000
#define MSR_DEFAULT            (MSR_SPE | MSR_WE | MSR_CE | MSR_ME)

/* Core 2 exception vectors initialization */
    .align      2
__iv2init:
    /* IVPR initialization.*/
    e_lis      %r3, HI(__ivpr2_base__)
    e_or2i     %r3, LO(__ivpr2_base__)
    mtIVPR     %r3

    /* MSR initialization.*/
    e_lis      %r3, HI(MSR_DEFAULT)
    e_or2i     %r3, LO(MSR_DEFAULT)
    mtMSR      %r3
```

The code above refers to the core_2. Similarly the same initialization can be done for all the other CPUs, referring to the corresponding base address for the Interrupt Vector, defined into the application linker script as well.

2.2.1.8 Relocating image from flash into RAM (optional)

Optionally, the application image stored into the read-only flash partition could be relocated into the R/W System SRAM to speed-up later access to code and data. So, with the exception of the .boot0 section, the whole flash image can be copied from flash to RAM. A properly configured linker script, will define some variable to identify the start, end and size of the image. This procedure could slightly slow down the boot step, but it will guarantee that the whole application data and code will later stay in RAM and benefit from increased R/W accesses. Also, in a late initialization, the D-Cache could be used also to give better performances.

The code in charge of doing this relocation, will look like:

```

/* Image relocation in RAM */
        e_lis      %r4, HI(__ram_reloc_start__)
        e_or2i     %r4, LO(__ram_reloc_start__)
        e_lis      %r5, HI(__ram_reloc_dest__)
        e_or2i     %r5, LO(__ram_reloc_dest__)
        e_lis      %r6, HI(__ram_reloc_end__)
        e_or2i     %r6, LO(__ram_reloc_end__)

.reloop:
__NOVLE

        cmpl       cr0, %r4, %r6

__VLE

        se_bge     .relend
        se_lwz     %r7, 0(%r4)
        se_addi    %r4, 4
        se_stw     %r7, 0(%r5)
        se_addi    %r5, 4
        se_b       .relloop

.relend:

```

Note: *This optional step is clearly executed only by the Boot CPU (Core_2).*

This is the last step of the system initialization. Booting CPU will continue execution to perform the next necessary steps for the C runtime setup, as said implemented in the crt0.s assembly file.

2.2.2 EABI “C runtime” initialization

According to Power Architecture EABI specification, to allow EABI compliant compilers to generate user application code that can work properly, some general purpose registers that play a special role during application execution must be initialized before starting execution of C code. These are the “special” usage registers:

- **GPR1:** Stack Frame pointer, set to the base of the stack
- **GPR2:** Small Data 2 pointer, set to the base of the ELF section .sdata2 plus 32 KB displacement
- **GPR13:** Small Data pointer, set to the base of the ELF section .sdata plus 32 KB displacement

First instruction of the crt0.s assembly file will do exactly this initialization. Code will look like the following one:

```

/* Stack setup */
e_lis      %r1, HI(__process_stack_end__)
e_or2i     %r1, LO(__process_stack_end__)
se_li      %r0, 0
e_stwu     %r0, -8(%r1)

/* Small sections registers initialization */
e_lis      %r2, HI(__sdata2_start__)
e_or2i     %r2, LO(__sdata2_start__)
e_lis      %r13, HI(__sdata_start__)
e_or2i     %r13, LO(__sdata_start__)

```

Symbols ‘__process_stack_end’, ‘__sdata2_start__’ and ‘__sdata_start__’ are appropriately defined into the linker script

For more details on the specification and usage of the Small Data section refer to the Power Architecture EABI specification (see [A.1: 1](#)).

2.2.3 Early Platform initialization

At this stage, the boot and crt code have performed all the basic system initialization. Boot CPU is ready to continue execution into the C code and continue with HW platform initialization. Obviously this task must be performed just once, so it is demanded to the Boot CPU. The reference implementation will provide a call back named ‘__early_init’ that must be implemented in some way. The reference software implements it into a C file named ‘clock.c’. We will describe in this section the main part of the platform initialization. As done previously the corresponding code is reported and commented.

2.2.3.1 Disable SWT device (optional step)

Both microcontrollers are equipped with some Software Watchdog Timers. The SWT can be used to detect software lockup if the peripherals is not properly and timely “feed”. For details on how to configure the various operations mode please refer to the RM. There is a SWT for each core. The reset values of the SWT are different for the Boot CPU core’s SWT where it is enabled by default. In the context of this application note, it could be required to disable the SWT to avoid any spurious and or unwanted reset. In order to enable the SWT, it needs to set the WEN (SWT_CR[31]) bit in the control register, that is read-only. In order to make them writeable, it is needed to write a specific sequence (0xc520, 0xd928) into the SWT Service Register (SWT_SR), specifically bits WSC[16:31].

An example code that could be used to disable all the SWT might look like:

```

/* SWTs disabled.*/
SWT_0.SR.R      = 0xC520U;
SWT_0.SR.R      = 0xD928U;
SWT_0.CR.R      = 0xFF000002UL;
SWT_2.SR.R      = 0xC520U;
SWT_2.SR.R      = 0xD928U;
SWT_2.CR.R      = 0xFF000002UL;
SWT_3.SR.R      = 0xC520U;
SWT_3.SR.R      = 0xD928U;
SWT_3.CR.R      = 0xFF000002UL;

```

The code will clear bit WEN (SWT_CR[31]) to disable and set bit FRZ (SWT_CR[30]) to stop the internal counter when the device is entering in debug mode (i.e. when attaching with a JTAG port). This example is covering the three SWT available on the SPC5EC8x device. On SPC5NG8x there is another instance that is the SWT_1. Anyway, by default only the SWT_2 instance on both microcontrollers is the only one that is enabled by default at reset time.

2.2.3.2 SSCM Error handling configuration

In order to support application debug, it's a good practice to instruct the device to force an error (by generating a prefetch or data abort exception) in case of illegal access to off-platform peripherals. We can have bus abort for access to unused or not-existent peripheral or peripherals that are not controlled by SSCM (controlled by bit PAE SSCM_ERROR[14]), or also bus abort for illegal access to reserved addresses within the address space of a valid peripherals (controlled by bit RAE SSCM_ERROR[15]). Reference code might look like:

```

/* SSCM_ERROR register bits definitions */
#define SPC5_SSCM_ERROR_RAE      (1U << 0)
#define SPC5_SSCM_ERROR_PAE      (1U << 1)

/* SSCM_ERROR register initialization */
#define SPC5_SSCM_ERROR_INIT      (SPC5_SSCM_ERROR_PAE |
SPC5_SSCM_ERROR_RAE)

/* SSCM initialization from configuration data.*/
SSCM.ERROR.R      = SPC5_SSCM_ERROR_INIT;

```

2.2.3.3 Reset sources clearing

It's a good practice to clear all information related to reset sources. The modules involved are the Reset Generation Module (MC_RGM) and the System Status and Configuration Module (SSCM). In particular it is needed to clear the RGM Destructive Event Status Registers (RGM_DES) and the Functional Event Status Register (RGM_FES). Both registers are cleared by writing '1'.

```

/* RGM errors clearing.*/
MC_RGM.FES.R      = 0xFFFFFU;
MC_RGM.DES.R      = 0xFFFFFU;

```

2.2.3.4 Clocks configuration

The next main step is concerned with the configuration of clock sources, clock tree and PLLs.

Regarding the clock sources, both devices provide some flexibility in order to choose the best source according to use case and/or peripherals constraints. Essentially there are five different sources:

- PLL0 (PLL0_PHI)
- PLL1 (PLL1_PHI)
- External oscillator (XOSC)
- External clock (EXTAL by-pass)
- Internal RC Oscillator (IRCOSC)

Note: For a detailed understanding of the whole clock structure, you should refer to the device RM (see [A.1](#): 2, 3).

When the device is powered-on, the default configuration enable only the internal RC oscillator IRCOSC as previously mentioned, and it is the source during the boot-up phase. Also all various dividers are set to '1' (in HSM module it is configured to provide the max frequency). This is not clearly a configuration that is able to provide the best clock frequency to the various peripherals. Hence the boot software has the responsibility to configure the required clock sources, initialize all the AUX selectors and dividers in order to provide the needed clock frequencies to the various peripherals and to configure the two PLLs.

2.2.3.4.1 Oscillator by-pass (optional step)

Depending on the board configuration, it is possible to by-pass the XOSC circuit and use an external crystal connected to the EXTAL pins. This configuration is managed by the MC_ME module through the XOSC digital interface. The XOSC control register (XOSC_CTL) is implemented for this purpose: bit OSCBYP (XOSC_CLT[0]) can be set by software only to by-pass the XOSC. It can be cleared only by a reset. The code above from the reference software corresponds to the oscillator by-pass. Clearly this is an optional setup.

```

#if (SPC5_OSC_BYPASS == TRUE)

/* If the board is equipped with an oscillator instead of a crystal then
the

bypass must be activated.*/

OSC40M_DIG.CTL.B.OSCBYP = TRUE;

#endif /* SPC5_OSC_BYPASS */

```

2.2.3.4.2 Clock tree configuration

Through the Clock Generation Module (MC_CGM) it is possible to fully configure the clock tree by selecting the needed clock sources and output dividers. The settings are applied by means of the Auxiliary Clock Selectors, through different sets of registers: DCx (Divider x

Configuration Register), SC (Select Control Register). Refer to the corresponding RM for the full description of the MC_CGM registers. Providing a full configuration for all clocks is out of the scope of this document, also because it's strongly dependent on the final user application requirements. The example reference code provides some basic configuration.

```
/* Setting the system dividers to their final values.*/

MC_CGM.SC_DC0.R      = SPC5_CGM_SC_DC0_BITS;
MC_CGM.SC_DC1.R      = SPC5_CGM_SC_DC1_BITS;
MC_CGM.SC_DC2.R      = SPC5_CGM_SC_DC2_BITS;
MC_CGM.SC_DC3.R      = SPC5_CGM_SC_DC3_BITS;
MC_CGM.SC_DC4.R      = SPC5_CGM_SC_DC4_BITS;

/* Setting the auxiliary dividers to their final values.*/

MC_CGM.AC0_DC0.R      = SPC5_CGM_AC0_DC0_BITS;
MC_CGM.AC0_DC1.R      = SPC5_CGM_AC0_DC1_BITS;
MC_CGM.AC6_DC0.R      = SPC5_CGM_AC6_DC0_BITS;
MC_CGM.AC9_DC0.R      = SPC5_CGM_AC9_DC0_BITS;
MC_CGM.AC12_DC0.R     = SPC5_CGM_AC12_DC0_BITS;
MC_CGM.AC12_DC1.R     = SPC5_CGM_AC12_DC1_BITS;
MC_CGM.AC12_DC2.R     = SPC5_CGM_AC12_DC2_BITS;
MC_CGM.AC12_DC3.R     = SPC5_CGM_AC12_DC3_BITS;
MC_CGM.AC12_DC4.R     = SPC5_CGM_AC12_DC4_BITS;

/* Setting the clock selectors to their final sources.*/

MC_CGM.AC0_SC.R       = SPC5_CGM_AC0_SC_BITS;
MC_CGM.AC3_SC.R       = SPC5_CGM_AC3_SC_BITS;
MC_CGM.AC4_SC.R       = SPC5_CGM_AC4_SC_BITS;
MC_CGM.AC6_SC.R       = SPC5_CGM_AC6_SC_BITS;
MC_CGM.AC9_SC.R       = SPC5_CGM_AC9_SC_BITS;
MC_CGM.AC12_SC.R      = SPC5_CGM_AC12_SC_BITS;
```

2.2.3.4.3 Enabling XOSC

The next step requires to enable the alternative clock source that is the XOSC. This configuration is controlled by the Module Entry block (MC_ME). The corresponding register is the ME_DRUN_MC (DRUN Mode Configuration) that controls the behavior of the system when in DRUN execution mode.

It is needed to set bit XOSCON (ME-DRUN_MC[26]) in order to enable the External Crystal Oscillator. Example reference code is as above:

```

/* ME_MC registers bits definitions */

#define SPC5_ME_MC_SYSCLK(n)          (((uint32_t)(n)) << 0)
#define SPC5_ME_MC_SYSCLK_IRC        SPC5_ME_MC_SYSCLK(0)
#define SPC5_ME_MC_SYSCLK_XOSC       SPC5_ME_MC_SYSCLK(1)
#define SPC5_ME_MC_IRCON              (1UL << 4)
#define SPC5_ME_MC_XOSCON             (1UL << 5)
#define SPC5_ME_MC_FLAON(n)          (((uint32_t)(n)) << 16)
#define SPC5_ME_MC_FLAON_NORMAL      SPC5_ME_MC_FLAON(3)
#define SPC5_ME_MC_MVRON              (1UL << 20)

/* Enables the XOSC in order to check its functionality before proceeding
   with the initialization.*/

MC_ME.DRUN_MC.R      = SPC5_ME_MC_SYSCLK_IRC | SPC5_ME_MC_IRCON |
                        SPC5_ME_MC_XOSCON | SPC5_ME_MC_FLAON_NORMAL |
                        SPC5_ME_MC_MVRON;

```

The bit to be set is represented by the macro SPC5_ME_MC_XOSCON. All the others bits are to preserve the default value (not only the bits are writeable, but access registers is word-size)

Note: *The system is expected to be in DRUN mode at the end of the reset procedure.*

2.2.3.4.4 PLL configuration

Last part of a minimal clock configuration concerns the configuration of the two PLLs. Shortly, the PLL topology is cascade, where PLL0 is provided with two outputs: PHI output is providing the pll0_clkout signal directly, and PHI1 is instead used as reference clock source for the PLL1, with just one output (pll1_clkout signal). The configuration of the PLLs is implemented by a common digital interface PLLDIG that provides some control and status registers. The register that control the programming id PLLxDV consisting of PHI (and PHI1 for PLL0) output clock reduced frequency dividers, (RFDPHI and RFDPHI1 bits), the Input Clock Pre-dividers (PREDIV bits) and the Loop Multiplication Factor divider (MFD bits).

Note: *PREDIV and MFD must be changed while PLL is in power-down and not locked, while RFDPHI/RFDPHI1 can be changed while in normal mode, considering that changes will become effective after a disable/enable cycle.*

A basic configuration can be as below code:

```

/* PLLs initialization, the changes will have effect on mode switch.*/
PLLDIG.PLL0CR.R      = 0U;

PLLDIG.PLL0DV.R      = SPC5_PLL0_DV_RFDPHI1(SPC5_PLL0_RFDPHI1_VALUE) |
                        SPC5_PLL0_DV_RFDPHI(SPC5_PLL0_RFDPHI_VALUE) |
                        SPC5_PLL0_DV_PREDIV(SPC5_PLL0_PREDIV_VALUE) |
                        SPC5_PLL0_DV_MFD(SPC5_PLL0_MFD_VALUE);

PLLDIG.PLL1CR.R      = 0U;

PLLDIG.PLL1DV.R      = SPC5_PLL1_DV_RFDPHI(SPC5_PLL1_RFDPHI_VALUE) |
                        SPC5_PLL1_DV_MFD(SPC5_PLL1_MFD_VALUE);

```

2.2.3.5 Chip mode of operation setup

The microcontroller can execute in different modes of operation to give flexibility to the application to implement different use cases based on different runtime hardware configuration; thus it is possible to implement application profiles with different characteristic in terms of power consumption and processing needs.

As soon as the system will exit from the reset, it will always enter in a so called DRUN mode. There are also other systems modes (RESET, SAFE and TEST) and user mode (RUN{0,3}, HALT0, STOP0, and STADBY0) that application can switch into to implement different use cases in terms of power consumption and processing needs. These modes can be also categorized into an orthogonal classification as “running” mode and “non running” mode. The [Table 6](#) will help you to understand the classification.

Table 6. Chip modes of operation

	Running	Not Running
System	DRUN TEST SAFE	RESET
User	RUN0 RUN1 RUN2 RUN3	HALT0 STOP0 STANDBY0

For a detailed description on the chip modes of operation, please refer to the related RM.

The execution modes are controlled by the Module Entry block (MC_ME). The whole configuration of the system and peripherals is realized through different levels:

1. Define which chip modes are allowed for the micro (by Mode Enable register ME_ME)
2. For each modes, define the overall system configuration in terms of power level, I/O output, voltage regulator, Flash status, PLLs status, XOSC, IRCOSC and system clock (by Mode Configuration register, ME_<mode>_MC)
3. For each peripherals, define which RUN and LP (low power) configuration/domain it belongs to (by Peripheral Control registers ME_PCTLn). There is a PCTLn register

associated to each IP/interface and there are up to 8 Peripheral RUN configuration and Peripheral Low-Power configuration.

4. For each of the 8 RUN configurations, define if the peripheral belonging to the configuration is active or frozen with the clock gated into the specific-running chip mode (by Run Peripheral Configuration registers, ME_RUN_PC{0,7})
5. For each of the 8 LP configurations, define if the peripheral belonging to the configuration is active or frozen with the clock gated into the specific not-running chip mode (by Low-Power Peripheral Configuration registers, ME_LP_PC{0,7})

Clearly the ME module gives an high flexibility and granularity by mapping the peripherals in different configurations, at the different chip modes allowing the user application to easily achieve its functional and power requirements.

The reference code provides a basic implementation that is described here thereafter:

1. All chip modes are allowed
2. Chip mode configurations are:
 - a) SAFE: hw reset value
 - b) DRUN: hw reset value plus PPL0, PLL1, XOSC switched ON, PLL1 as system clock
 - c) RUN0: same as DRUN
 - d) RUN1, RUN2, RUN3: hw reset value
 - e) HALT0: hw reset value
 - f) STOP0: hw reset value
 - g) STANDBY0: hw reset value
3. For each peripherals, keep the default grouping (RUN_PC0 and LP_PCO)
4. RUN peripheral configurations
 - a) RUN_PC0: "Never run" mode, that is peripherals are OFF in all modes
 - b) RUN_PC1: "Always run" mode, that is peripherals are ON in all modes
 - c) RUN_PC2: "Normal mode run", that is peripherals are ON in DRUN and all RUNx modes
 - d) RUN_PC3-7: "Application specific", available to implement custom configuration (currently as RUN_PC2)
5. LP peripheral configurations
 - a) LP_PC0: "Never run" mode, that is all peripherals are OFF in all not-run modes
 - b) LP_PC1: "Always run", that is all peripherals are ON in all not-run modes
 - c) LP_PC2: "Halt only", peripherals ON in HALT0 mode only
 - d) LP_PC3: "Stop only", peripherals are ON in STOP0 mode only
 - e) LP_PC4-7: "Application specific", available to implement customer configuration (currently peripherals are ON in HALT0 and STOP0 modes)

```

/* Run modes initialization, note writes to the MC registers are verified
   by a protection mechanism, the operation success is verified at the
   end of the sequence.*/

MC_ME.IS.R          = 8U;                      /* Resetting I_ICONF status.*/
MC_ME.ME.R          = SPC5_ME_ME_BITS;
MC_ME.SAFE_MC.R      = SPC5_ME_SAFE_MC_BITS;
MC_ME.DRUN_MC.R      = SPC5_ME_DRUN_MC_BITS;
MC_ME.RUN_MC[0].R    = SPC5_ME_RUN0_MC_BITS;
MC_ME.RUN_MC[1].R    = SPC5_ME_RUN1_MC_BITS;
MC_ME.RUN_MC[2].R    = SPC5_ME_RUN2_MC_BITS;
MC_ME.RUN_MC[3].R    = SPC5_ME_RUN3_MC_BITS;
MC_ME.HALT0_MC.R     = SPC5_ME_HALT0_MC_BITS;
MC_ME.STOP0_MC.R     = SPC5_ME_STOP0_MC_BITS;
MC_ME.STANDBY0_MC.R  = SPC5_ME_STANDBY0_MC_BITS;
if ((MC_ME.IS.B.I_ICONF & 1U) == 1U) { /* Configuration rejected.*/
    SPC5_CLOCK_FAILURE_HOOK();}

/* Peripherals run and low power modes initialization.*/
MC_ME.RUN_PC[0].R    = SPC5_ME_RUN_PC0_BITS;
MC_ME.RUN_PC[1].R    = SPC5_ME_RUN_PC1_BITS;
MC_ME.RUN_PC[2].R    = SPC5_ME_RUN_PC2_BITS;
MC_ME.RUN_PC[3].R    = SPC5_ME_RUN_PC3_BITS;
MC_ME.RUN_PC[4].R    = SPC5_ME_RUN_PC4_BITS;
MC_ME.RUN_PC[5].R    = SPC5_ME_RUN_PC5_BITS;
MC_ME.RUN_PC[6].R    = SPC5_ME_RUN_PC6_BITS;
MC_ME.RUN_PC[7].R    = SPC5_ME_RUN_PC7_BITS;
MC_ME.LP_PC[0].R     = SPC5_ME_LP_PC0_BITS;
MC_ME.LP_PC[1].R     = SPC5_ME_LP_PC1_BITS;
MC_ME.LP_PC[2].R     = SPC5_ME_LP_PC2_BITS;
MC_ME.LP_PC[3].R     = SPC5_ME_LP_PC3_BITS;
MC_ME.LP_PC[4].R     = SPC5_ME_LP_PC4_BITS;
MC_ME.LP_PC[5].R     = SPC5_ME_LP_PC5_BITS;
MC_ME.LP_PC[6].R     = SPC5_ME_LP_PC6_BITS;
MC_ME.LP_PC[7].R     = SPC5_ME_LP_PC7_BITS;

/* Switches again to DRUN mode (current mode) in order to update the settings.*/
if (SPC5_SetRunMode(SPC5_RUNMODE_DRUN) == CLOCK_FAILED) {
    SPC5_CLOCK_FAILURE_HOOK();}

```

2.2.4 DATA sections handling

At this stage all the early platform initialization steps have been completed. Just before the Boot CPU could jump to user application code for executing the application, the “C runtime” code needs to cope with the DATA sections of the application binary.

2.2.4.1 Clearing BSS sections

The BSS sections need to be zero-ed as required by the C language semantic. A simple loop for clearing the BSS in memory will look like:

```
/* BSS clearing */
e_lis      %r4, HI(__bss_start__)
e_or2i     %r4, LO(__bss_start__)
e_lis      %r5, HI(__bss_end__)
e_or2i     %r5, LO(__bss_end__)
se_li      %r7, 0
.bssloop:
    cml     cr0, %r4, %r5
    se_bge  .bssend
    se_stw  %r7, 0(%r4)
    se_addi %r4, 4
    se_b    .bssloop
.bssend:
```

2.2.4.2 Copying initialized DATA section from flash to RAM

The DATA sections containing all the initialized data must be copied from the application image into the System RAM. The example code will consider a boot configuration where the application is already loaded in memory by some debug tools, thus not executing from flash. In that case copying DATA sections is not a needed action. The part of the reference code where this copy is done is as below:

Late platform initialization and main execution

```
#if !BOOT_LOAD_IN_RAM
    /* DATA initialization */
    e_lis      %r4, HI(__romdata_start__)
    e_or2i     %r4, LO(__romdata_start__)
    e_lis      %r5, HI(__data_start__)
    e_or2i     %r5, LO(__data_start__)
    e_lis      %r6, HI(__data_end__)
    e_or2i     %r6, LO(__data_end__)
.dataloop:
    cmpl       cr0, %r5, %r6
    se_bge     .dataend
    se_lwz     %r7, 0(%r4)
    se_addi    %r4, 4
    se_stw     %r7, 0(%r5)
    se_addi    %r5, 4
    se_b       .dataloop
.dataend:
#endif
```

2.2.5 Late platform initialization and main jump

All the basic tasks required to execute any application on the microcontrollers have been done. The reference code will provide an hook for the end application to execute any custom tasks for further configuration and/or platform tuning and optimization. The end-user will require eventually to implement such hook “__late_init”. After that the code will branch to the ‘main’ C symbol. Eventually an exit handler could be implemented for cleanup purpose (application provide a fake empty implementation). The corresponding code is as below:

```

/*
 * Late initialization.
 */
e_bl      __late_init
/*
 * Main program invocation.
 */
e_bl      main
e_b       _main_exit_handler

/*
 * Default main exit code, infinite loop.
 */
.weak     _main_exit_handler
.globl    _main_exit_handler
.type     _main_exit_handler, @function
_main_exit_handler:
e_b       _main_exit_handler

```

2.3 Multicore system startup

As described in [Section 1.3: MCU Cores features](#) the SPC58xCx/SPC58xG8x microcontrollers are in a multicore configuration; excluding the HSM and the Checker code, we have two general purpose CPUs into SPC58xCx and three general purpose CPUs into SPC58xG8x. In [Section 2.1: Start-up flow \(high-level view\)](#) the start-up flow for a single-core configuration has been described. In this section it will be described the multi-core scenario with the different possibility that can be implemented.

2.3.1 Multi-core boot (by CPU)

In this case, the actions to boot the secondary cores are taken by the Boot CPU that is the only one woken-up at the end of the reset procedure. This is the only way to have a multi-core boot when the BAF code is not executed (SSCM will only dispatch reset vector to Boot CPU).

Boot CPU will perform usual system initialization (core and platform) then it will wake-up the other secondary cores, providing dedicated entry point to each core, through the Module Entry module (MC_ME).

The code running on the secondary cores will perform all the mandatory initialization steps as executed by main core and will skip the platform setup (it must be done once by the Boot CPU only). All the secondary cores will then start and executing the user application stuff that can be completely different by the task executed by the main core. The split of the user

tasks across the different CPUs is tightly coupled and dependent by the complete end user application.

2.3.1.1 Module Entry support for multi-core boot

The MC_ME not only controls chip mode an, its configuration and peripheral run configuration for the various run and non-run modes, but it is also provided with clock gating and boot address controls for the secondary cores, through a set of registers:

- Core Control Register N (ME_CCTLn): it controls if the specified core is disabled (frozen with its clock gated) or running for each chip mode (run and non-run modes). Note that all cores are always disabled in the non-run modes (STANDBY0, STOP0 and HALT0); conversely the Boot CPU of the microcontroller (core_2) is by default enabled and running in all the run modes, both system (DRUN, SAFE, TEST) and user (RUN0..3).
- Core Address Register N (ME_CADDRn): the lowest bits [0:29] provides the boot address of the corresponding core at the next exit from a reset, while the bit 31 [RMC, Reset on Mode Change] determines whether the core should be reset (RMC: 1) or not (RMC: 0) at the next mode change where the target mode includes the specified core in the running configuration.

Thus, in order to trigger a software controlled boot of one of the secondary core, the following steps must be executed:

1. Identify a target mode for the next transition (it could be the same mode as the current one, i.e. DRUN). Note that the target mode must be allowed in ME_ME register
2. Make sure that the specified CPU is enabled for the target mode (ME_CCTLx)
3. Setup the boot address and enable the RMC bit (ME_CADDRx)
4. Issue a mode transition for the chosen target mode. This is achieved in multiple steps
 - a) Write key value (0x5AF0) in MC_MCTL[KEY] bits and chip mode value in MC_MCTL[TARGET_MODE] bits
 - b) Write it again but using the inverted key value (0xA50F)
 - c) Check that there are not pending mode transition by inspecting ME_GS[S_MTRANS] bit
 - d) Check that the current mode is the expected one by inspecting ME_GS[S_CURRENT_MODE] bits.

See [Table 7](#) for a summary view of the mapping between the ME_CADDRx/ME_CCTLx registers and the CPUs available on the SPC58xCx/SPC58xG8x family microcontrollers.

Table 7. ME_CADDRx/ME_CCTLx registers mapping

Register id X	SPC58ECx	SPC5NG8x
1	Core_2	Core_2
2	Core_0	Core_0
3	—	Core_1
4	HSM	HSM

2.3.2 Multi-core boot (by BAF)

In case of the BAF code is executed by the Boot CPU stimulated by the SSCM, depending on the configuration written into the Boot header, more than one CPUs can be woken-up all together at the end of the BAF code execution and start executing user code at the different vector addresses provided into the header. See [Table 4](#) for the description of the Boot Record searched by the BAF code.

The work split is exactly the same as in the first case: Boot CPU will usually perform core and platform initialization, while the secondary cores will execute only core specific initialization. Then all the cores will start executing the end user application. Some synchronization will need to be implemented in user application to ensure that the system is in the proper state before executing its duties (this strongly depend on the tasks that are performed by the CPU).

Appendix A Further Information

A.1 Reference documents

1. *e200z4 Power Architecture® Core Reference Manuals*
2. *SPC584Cx/SPC58EC8x 32-bit MCU family built on the Power Architecture® for automotive body electronics applications* (RM0407, Doc ID 028117)
3. *SPC58NE84x/SPC58xG84x 32-bit Power Architecture® microcontroller for automotive ASILD applications* (RM0391, Doc ID 027214)

A.2 Acronyms and abbreviations

Table 8. Acronyms and abbreviations

Acronyms	Meaning
BAM	Boot Assist Module
BAF	Boot Assist Flash
BTB	Branch Target Buffer
CPU	Central Processing Unit
DCache	Data Cache
DCF	Device Configuration Format
DMEM	Data Memory (internal to the core)
HSM	Hardware Security Module
ECC	Error Correcting Code
GHS	Green Hills (Compiler)
ICache	Instruction Cache
IMEM	Instruction Memory (internal to the core)
LSM	Lock Step Mode
MC_CGM	Clock Generation Module
MC_ME	Mode Entry Module
RGM	Reset Generation Module
MPU	Memory Protection Unit
PMC	Power Management Controller
PLL	Phase Locked Loop

Revision history

Table 9. Document revision history

Date	Revision	Changes
24-Feb-2017	1	Initial release.
04-Sep-2017	2	In Section 2.2.1.2: Clearing of all core registers , added paragraph "It's also useful to cleanup CPU register..." and relative code. In Section 2.2.1.6: Caches initialization , updated "se_andi" with "e_and2i." code. Removed "ST Restricted" watermark.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2017 STMicroelectronics – All rights reserved