AN4664
Application note

SPC56ELxx Automotive MCU multicore architectures and getting started

Introduction

This document provides an introduction to the world of multi-core MCU architectures and programming and ST associated solutions.

The requirements in terms of performance of embedded systems are always rising. To fulfill the expectation of customers it is necessary to increase also performance of microcontrollers. In order to fulfill such requirements designers have two choices, either adding different cores to the microcontrollers or increasing performance of monolithic processors.

Multiple cores can run multiple instructions at the same time increasing overall performance. Multi-core devices also allow higher performance at lower energy. It can be a great factor in mobile devices that operate on batteries. Since each core in multi-core is generally more energy-efficient, the chip becomes more efficient than having a single large monolithic core.

Multi-core solution seems to be more effective for processor designers and most of the companies follow this strategy.

Assuming that the die can fit into the package, physically, the multi-core device designs require much less printed circuit board (PCB) space than do multi-chip symmetric multiprocessing (SMP) designs. Also, a dual-core processor uses slightly less power than two coupled single-core processors, mainly because of the decreased power required to drive signals external to the chip.

Maximizing the utilization of the computing resources provided by multi-core processors requires adjustments both to the operating system (OS) support and to existing application software. Also, the ability of multi-core processors to increase application performance depends on the usage of multiple threads within applications.

Some architectures use one core design repeated consistently (“homogeneous”), while other ones use a mixture of different cores, each optimized for a different, “heterogeneous” role.

In addition to the increase in terms of computational performance, multi-core architectures help the development of safety related applications in which, for example, a core works as main core and the other as checker.

The basic idea is that to reach some of the safety requirements, two cores can execute the same calculation on the same inputs and compare their outputs. This kind of dual-core processors are commonly used in automotive safety relevant applications.

In such way it’s possible to reach a certain separation between main safety relevant tasks and monitor tasks. The software running on main core does not interfere on checker core and vice versa

Contents

List of tables

List of figures

1 Multi-core architectures and programming

The term “Multicore” represents a device that has multiple CPUs interconnected over a chip-level bus embedded in the same silicon die. Multicore processors deliver greater computing power through concurrency, offer greater system density, and then can run at lower clock speeds than uniprocessor chips.

Let’s focus now on dual-core microcontrollers as they are the basics from Multicore processors.

From the hardware point of view the basic classification is homogeneous (SMP) and heterogeneous (AMP) multiprocessing.

Homogenous (SMP) – same cores are embedded in the same device

An example for homogenous multiprocessing is the SPC56AP60, which embeds two e200z0 cores, or SPC56EL60, which embeds two e200z4 cores.

Heterogeneous (AMP) – different cores are embedded in the same device

An example for heterogeneous multiprocessing is SPC56EC7 which embeds a master core e200z4 and a slave core e200z0.

SPC56Exx family has been designed for Body applications in Automotive. It’s focused on power saving during idle state and therefore the second core is there mostly for low-power modes handling and doesn’t need to be as powerful as main core.

From the software point of view:

SMP – functional tasks usually run on different cores

AMP – functional tasks usually run to a single core[a]

The main benefits from multi-core compared to monolithic single chip processor are:

- Less cost
- Less space
- Less power
- Less heating

1.1 Symmetric vs. Asymmetric Multi-Processing

Another classification is between symmetric and asymmetric multiprocessing.

1.1.1 Asymmetric multiprocessing

Asymmetric multiprocessing represents different cores, possible instruction sets or different operating systems, which may not have access to the same resources.

Tasks, which run on different cores, use messages for communication. User must either implement a proprietary communication scheme or choose two OSs that share a common infrastructure (likely IP based) for inter-processor communication.

a. The slave core is normally used for secondary tasks as monitoring, low-power modes, IO management and so on.

To help avoiding resource conflicts, the OSs should also provide standardized mechanisms for accessing shared hardware components. Normally, this resource allocation occurs statically during boot time.

“Resource allocation” includes physical memory allocation, peripheral usage and interrupt handling.

Allocating the resources dynamically may entail complex coordination between the multiple cores.

In an AMP system, a task usually runs on the same CPU, even when other CPUs run idle. As a result, one CPU can end up being under or over-utilized. To address the problem, the system could allow applications to migrate dynamically from CPU to another.

In this way it can involve a complex check-pointing of state information or a possible service interruption. As the application is stopped on one CPU and restarted on another. Also, such migration is difficult, if not impossible, if the CPUs run different OSs.

1.1.2 Symmetric multiprocessing

It consists of identical (homogenous) cores with common shared memory and usually one operating system. A symmetric multiprocessing system has identical access to all system resources (e.g. memory, disks, UARTs, communication controllers).

The OS has insight into all system elements at all times. It can allocate resources on the multiple CPUs with little or no input from the application designer.

1.2 Lock-step and Decoupled mode

Multi core architectures can be splitted on Lock step mode (LSM) or Decoupled mode (DPM).

1.2.1 Lock-step mode

In lock step mode the same instruction is executed by both cores in synchronization[b].

This working mode is specifically designed to simplify the development of safety related application. In lock-step it’s possible to reach high integrity level, e.g. ASIL D, with minimal software overhead.

SPC56Elxx, i.e. SPC56EL60, working in LSM has been certified by external company to meet the ISO 26262 ASIL D requirements.

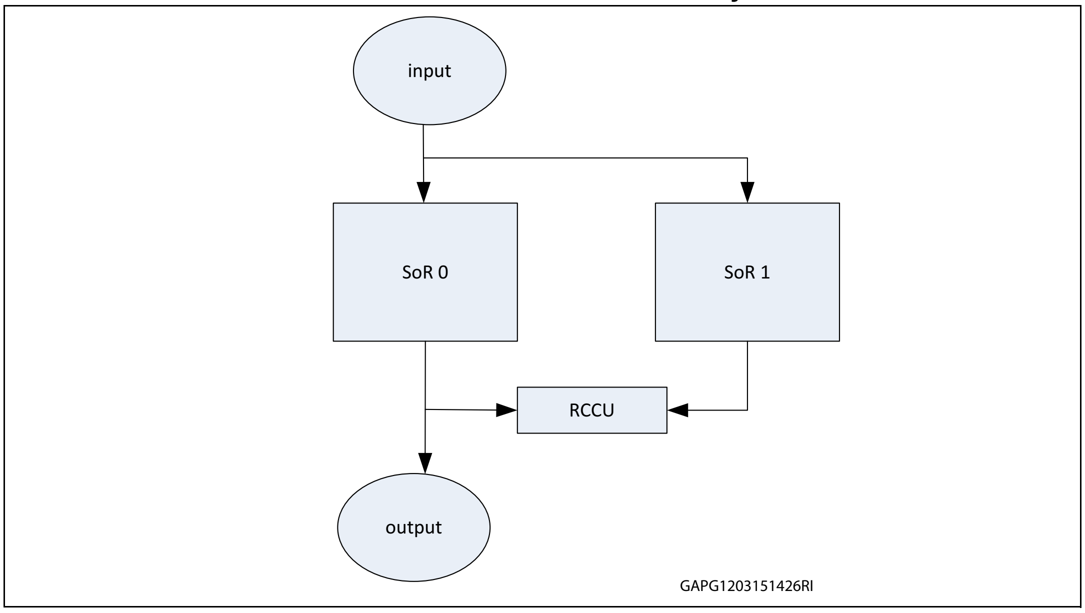
In this mode the Sphere of Replication (SoR) plays a major role. It contains all hardware elements which have been replicated for safety reasons resulting in the sphere being a collection of hardware pairs. Each member of such a pair executes the same operations or transactions as its partner resulting in lock step behavior.

The compliance with this expected behavior is checked only on the boundary of the SoR, minimizing checker effort.

b. In delayed lock-step architecture a fixed delay, e.g. 2 clock cycles, between the execution of the instruction by the first core and the second one is inserted.

This boundary check is based on a modified version of the fault isolation concept. Fault isolation requires that a fault must not cause failures outside a marked area, in this case the SoR. A failure in the SoR, as long as it does not propagate to the outside of the SoR and potentially causes a fault there, does not influence the effective operability of the periphery (and so the ECU). Thus it cannot cause a hazard.

Figure 1. A fault is detected when it goes outside the SoR. Until the fault remains isolated inside the SoR can't cause any hazard



For example, an error in the ALU can cause wrong calculation results but as long as these results only influence core register values, they can't generate a hazard at system level. Also, propagation inside the SoR is of no immediate consequence, e.g. if the wrong register value is written to the INTC, this — in itself — is not change the overall behavior of the system.

But once the registers are written somewhere external or used as addresses, or once the badly changed interrupt triggers, this 'safeness' changes because the failure now propagates outside of the SoR.

The Redundancy Control Checker Units (RCCU) at the outputs of the SoR detects such propagating failures due to data on the external busses being inconsistent between both processing units.

The RCCUs detect, but not prevent, the propagation of a non-common cause failure at the point where the two redundant channels are merged into a single actuator or recipient.

The detection of the failure is forwarded to the external “world” which shall react to put system into a safe state.

From programmers’ perspective microcontroller in Lock-step mode behaves like single core controller with added RCCU. This means one linker file, no need to start second core and one continuous SRAM.

1.2.2 Decoupled mode

In this mode, each core, and connected modules, run independently from the other one and redundancy checkers (RCCU) are disabled or not present.

The DPM increased performances can be estimated in first approximation as about 1.6x the performance of the LS mode at the same frequency.

If DPM mode is selected, then core0 starts. This core (core0) can enable core1. If core1 is not enabled, it is not clocked. This can be beneficial if user only needs to use core0, because core1 is not consuming any current and the chip's overall current consumption is therefore reduced.

Most common use of device in DPM is to separate AUTOSAR software from complex drivers. To make a picture: AUTOSAR software is running on master core (core0) independently of complex drivers and time critical applications which runs on the second core (core1).

In this operating mode, the chip boots with Core_0 enabled and Core_1 disabled(c).

Software running on Core_0 can enable Core_1 at any time, and once core1 has been enabled, it cannot be disabled by software. While Core_1 is disabled, it is not clocked, thus minimizing the chip's overall current consumption during this time.

1.3 Multi core architectures in ST

ST dual core microcontroller devices are designed to work in one of two operating modes. User can choose a product offering either lock step mode or decoupled mode. There is also possibility to have microcontrollers that implements both of these modes

- Decoupled mode: SPC56AP60, SPC56EC7
- Lock step & Decoupled mode: SPC56EL60, SPC56EL70 and next generation of multi cores devices.

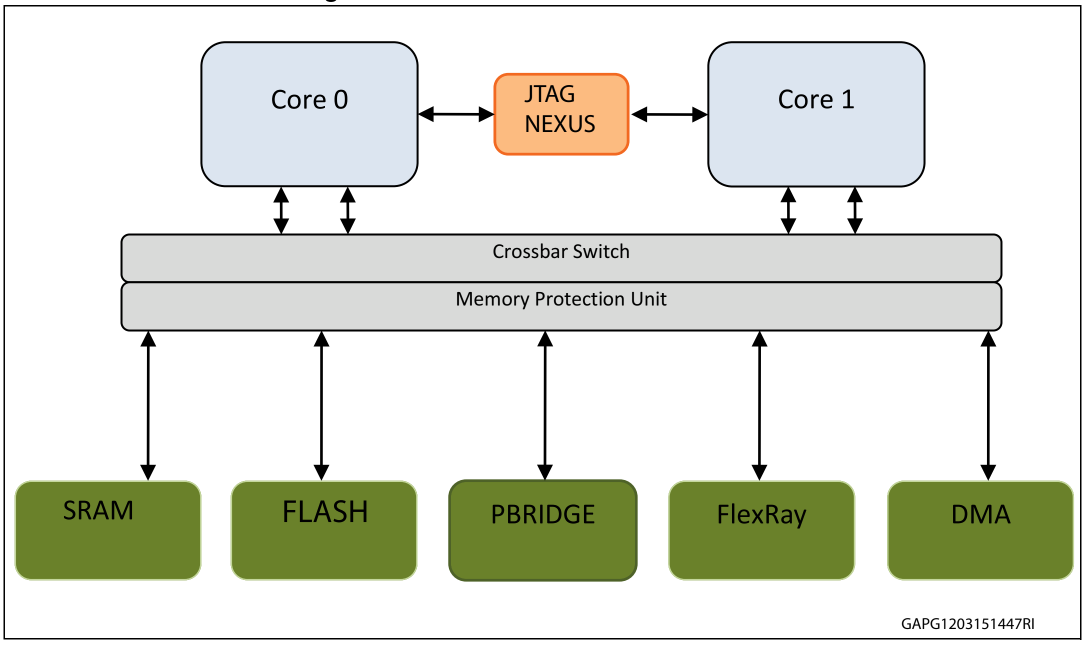
For the latter devices only one operating mode can be active at once. After the mode change one of the two modes is statically selected at power-up. Selected mode may be changed only going through a full power-on reset.

1.3.1 Architecture

Figure 2 is an example for Power architecture® dual-core.

c. This is not true for all reset events, in case of short external or short 'functional' reset, core1 is immediately enabled if it was enabled prior to the reset.

Figure 2. Power architecture dual-core



2 ST dual-core products

Table 1. ST’s Dual-core microcontrollers

Device	Cores	Operating modes	Class
SPC56EL60	z4 + z4	LSM + DPM	Chassis & safety
SPC56EL70	z4 + z4	LSM + DPM	Chassis & safety
SPC56AP60	z0 + z0	DPM	Chassis & safety
SPC56EC7	z4 + z0	DPM	Body

3 Getting started

3.1 Used microcontroller and tools

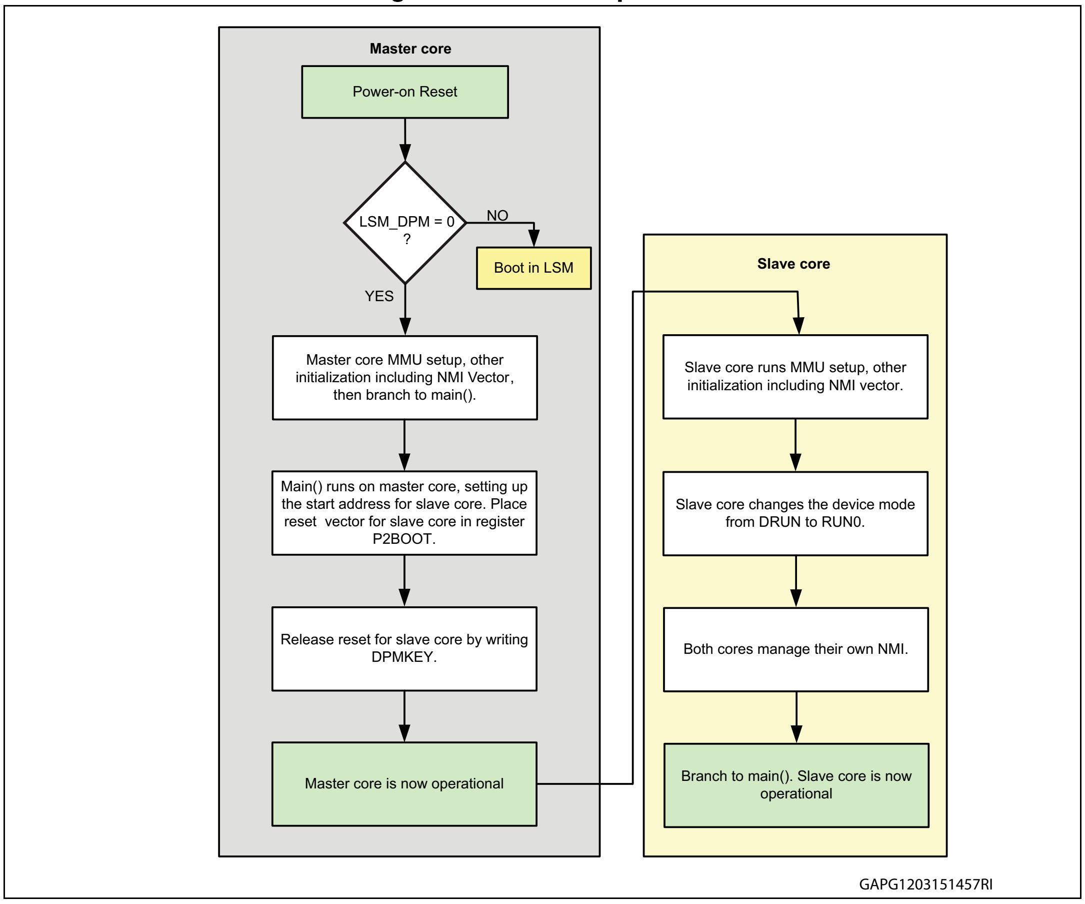
The presented application example is for dual core microcontroller in Decoupled mode. Demonstration is done for ST’s SPC56EL70 processor from chassis-safety family with 128 KB RAM and 2 MB Flash. The tools used are Lauterbach debugger and GreenHills compiler. The device chosen is SPC56ELxx because it combines DPM and LSM modes.

The example below describes step-by-step how to create a simple application for dual-core SPC56ELxx microcontroller using 2 linker files.

3.2 Entering DPM

Entering DPM implies a dual-core boot. The key concept to a dual-core boot is that it is nothing more than a typical single-core boot, except that it starts another single-core boot. The initialization of interrupts, stack, and other parameters needs to be performed on each core. In other words, it is a single-core boot performed twice.

Figure 3. DPM boot process


Master core
Power-on Reset
LSM_DPM = 0 ?
NO
Boot in LSM
YES
Master core MMU setup, other initialization including NMI Vector, then branch to main().
Main() runs on master core, setting up the start address for slave core. Place reset vector for slave core in register P2BOOT.
Release reset for slave core by writing DPMKEY.
Master core is now operational
Slave core
Slave core runs MMU setup, other initialization including NMI vector.
Slave core changes the device mode from DRUN to RUN0.
Both cores manage their own NMI.
Branch to main(). Slave core is now operational
GAPG1203151457RI

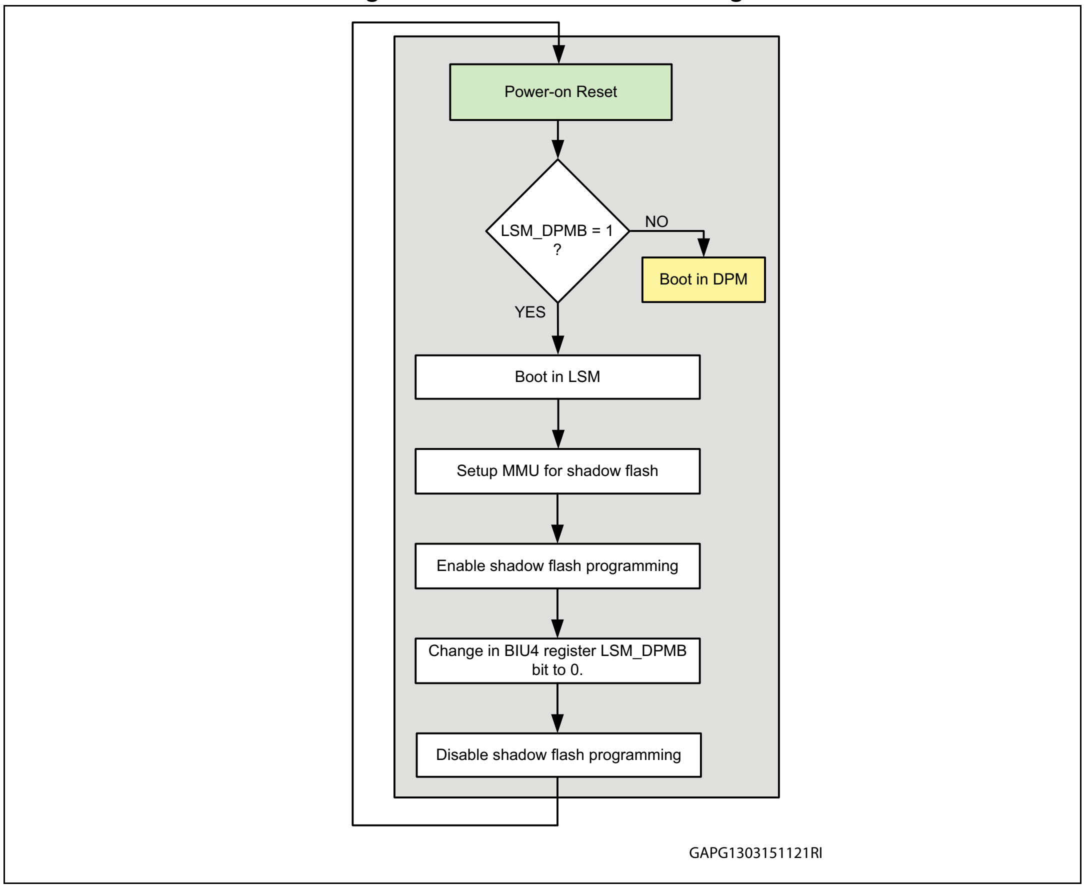
At power-on reset (POR), Core_0 (Master core) begins operation while Core_1 (Slave core) remains held in reset. At this time, Core_0 must initialize its set of peripherals, set up its environment (including the NMI routine), then branch to main. At this point, Core_0 is essentially fully operational. Now Core_0 provides the reset vector and writes the DMPKEY, thus releasing Core_1 from reset. Core_1 begins its execution. The first thing that it must do is initialize its set of peripherals and set up its environment, including its NMI routine. Core_0 then moves the chip from DRUN mode to RUN0 mode. Each core must service its own NMI routines. It is recommended to always use Core_0 to control the chip modes in order to avoid conflicting or inconsistent chip mode configurations and change requests.

3.2.1 Switching from LSM to DPM mode

Because SPC56ELxx microcontroller contains both LSM and DPM modes it is necessary to ensure that microcontroller is in DPM mode. By default SPC56ELxx comes from factory with LSM mode programmed. Therefore the mode must be switched to DPM in order to be able to start the second core in DPM mode. Below it is described how to change modes of SPC56ELxx.

The LSM and DPM mode switch is stored in the shadow sector of the Flash memory. This is a special sector of Flash memory where user options configuration bits are stored.

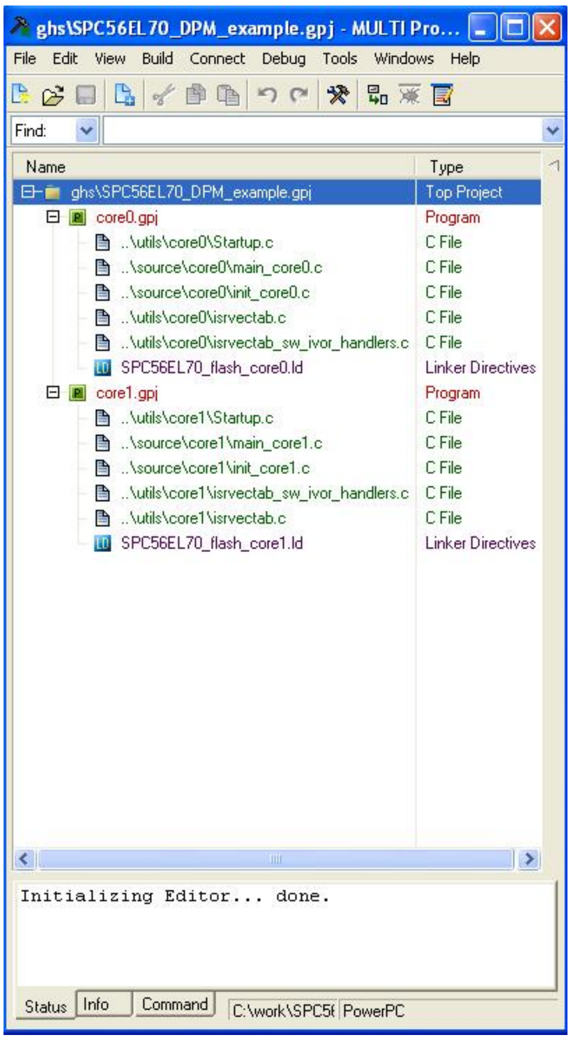
Figure 4. LSM to DPM switching


Power-on Reset
LSM_DPMB = 1 ?
NO
Boot in DPM
YES
Boot in LSM
Setup MMU for shadow flash
Enable shadow flash programming
Change in BIU4 register LSM_DPMB bit to 0.
Disable shadow flash programming
GAPG1303151121RI

4 Creating project in GreenHills compiler

First step is to create project in GreenHills compiler as shown on *Figure 5*.

Figure 5. Example project



Name	Type
ghs\SPC56EL70_DPM_example.gpj	Top Project
core0.gpj	Program
..\utils\core0\Startup.c	C File
..\source\core0\main_core0.c	C File
..\source\core0\init_core0.c	C File
..\utils\core0\isrvectab.c	C File
..\utils\core0\isrvectab_sw_ivor_handlers.c	C File
SPC56EL70_flash_core0.ld	Linker Directives
core1.gpj	Program
..\utils\core1\Startup.c	C File
..\source\core1\main_core1.c	C File
..\source\core1\init_core1.c	C File
..\utils\core1\isrvectab_sw_ivor_handlers.c	C File
..\utils\core1\isrvectab.c	C File
SPC56EL70_flash_core1.ld	Linker Directives

```
Initializing Editor... done.
```

Some examples with two separated programs (core0 and core1). Each program containing its own linker file, startup file, main and interrupt vector tables. As the cores are never started at once they are usually marked as core0 as a master core and core1 as slave core.

4.1 Startup

Startup is first code which is executed after power-on and reset phase. It must contain basic initialization for microcontroller. To be able to work with Flash and SRAM memory it is necessary to initialize MMU (memory management unit). Each core has its own MMU unit. Therefore initialization must be done twice. It is possible to configure MMUs differently for each core.

Figure 6. Basic start-up flow

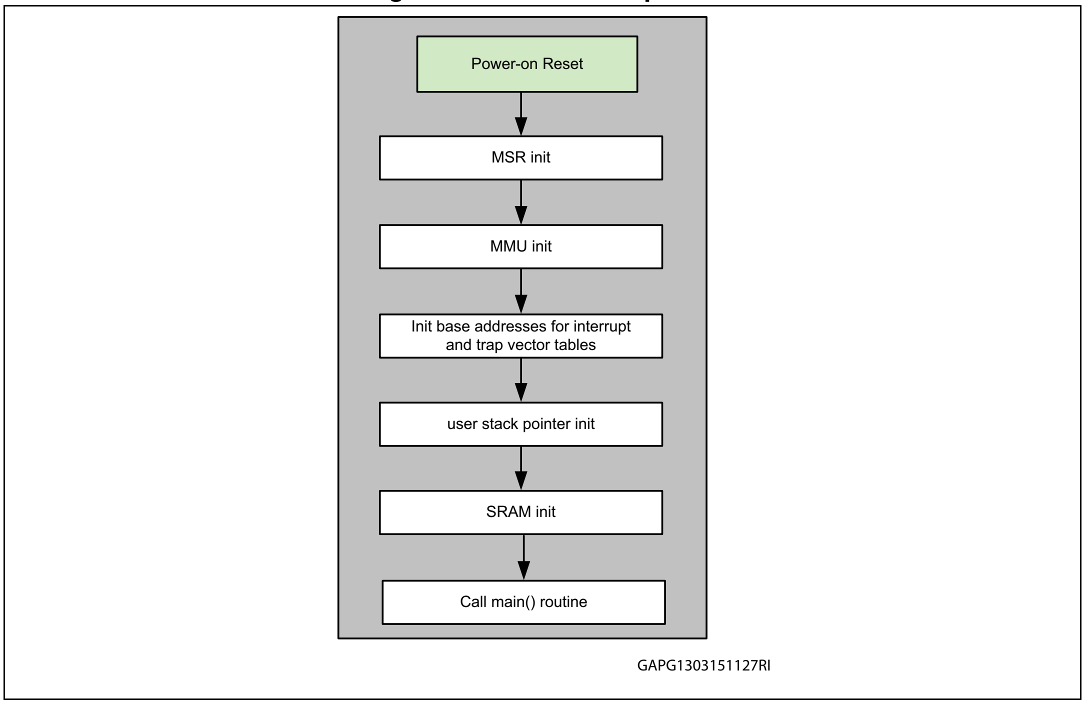
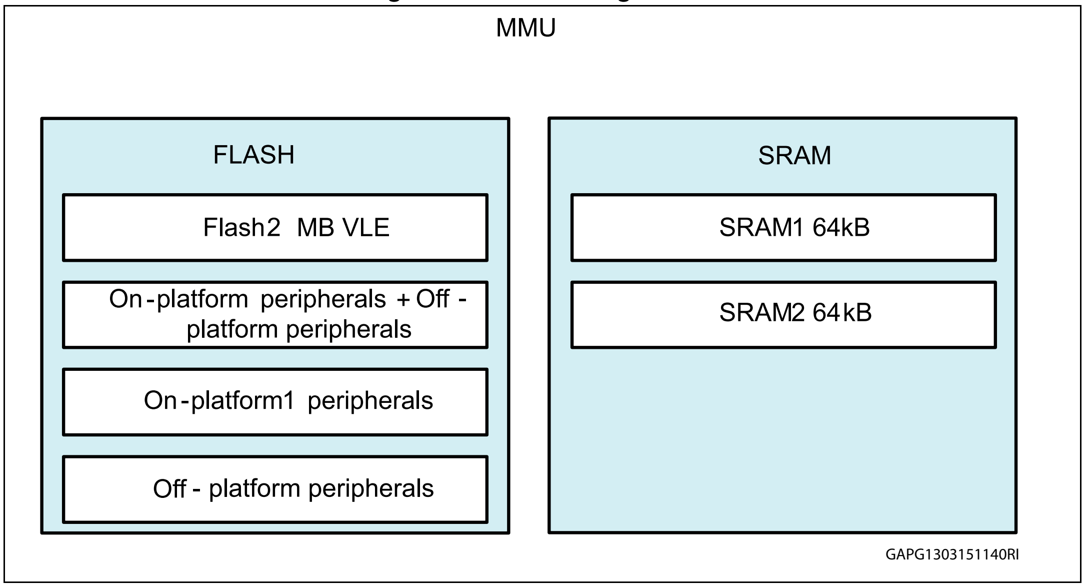


Figure 5 presents a very basic startup initialization for SPC56ELxx.

4.1.1 MMU initialization

Figure 6 shows all parts of Flash memory which can be initialized except test and shadow sector blocks. Usually SRAM1 is assigned to Master core (core0) and SRAM2 is assigned to Slave core (core1). But it is possible to initialize both SRAMs for both cores MMUs.

Figure 7. MMU configuration



Section Appendix A: Master core MMU initialization example shows an example of MMU initialization for master core. Compiler does not require any special optimization for dual-core processors.

4.1.2 Interrupt and trap vector tables

The start-up code shall initialize the base addresses for interrupt and trap vector tables. These base addresses are provided as configuration parameters or linker/locator setting.

4.1.3 Stack pointer initialization

The start-up code shall initialize the user stack pointer. The user stack pointer base address and the stack size are provided as configuration parameter or linker/locator setting.

4.1.4 SRAM init

The start-up code shall initialize a minimum amount of RAM in order to allow proper execution of the MCU driver services and the caller of these services.

Similar way is used for Slave core start-up initialization.

4.2 Linker files

Example 1 has 2 linker files. Therefore, the whole project seems like 2 independent programs. The only one link between those two programs at this moment is starting slave core from main function of master core.

There is also the possibility to use only 1 linker file. Then the sections must not overlap. For some Flashing devices it is easier to operate with only one linker file. This means there is only one output file.

The starting section of second core must be placed in different boot address vector than master core boot address. Usually master RCHW (reset configuration half-word) is placed in address 0x0. Slave core boot address depends on bootable sectors of microcontroller.

Example 1 linker script example for Master core:

Example 1

```
/* Master Core */
DEFAULTS
{
    stack_reserve = 4K
    heap_reserve = 2K

    /* SPC56xxL 1M Internal Flash */
    int_Flash_size = 2M
    /* SPC56xxL 128K Internal SRAM */
    int_ram_size = 64K
}

MEMORY
```

```
{
    /* Internal Flash RCW */
    Flash_rchw : ORIGIN = 0x00000000, LENGTH = 8
    int_Flash  : ORIGIN = 0x00000008, LENGTH = int_Flash_size - 8
    int_sram   : ORIGIN = 0x40000000, LENGTH = int_ram_size
 }

SECTIONS
{
    /*** ROM data ***/
    .rchw  NOCHECKSUM : { *(.rchw) } > Flash_rchw
    .start                           : > int_Flash
.
.
.
```

The linker script for Master Core contains RCHW section. The boot address vector is placed in this section at address 0x0.

Linker script example for Slave core:

```
/* Slave Core */
DEFAULTS
{
    stack_reserve = 4K
    heap_reserve = 2K

    /* SPC56xxL 1M Internal Flash */
    int_Flash_size = 2M
    /* SPC56xxL 128K Internal SRAM */
    int_ram_size = 64K
}

MEMORY
{
    int_Flash  : ORIGIN = 0x00008000, LENGTH = int_Flash_size
    int_sram   : ORIGIN = 0x50000000, LENGTH = int_ram_size

}
SECTIONS
{
    /*** ROM data ***/
    /* RCHW section is not declared for second core - NO RCHW!!! */
    .start                           : > int_Flash
```

The linker script for slave core does not contain RCHW section, because the boot vector address for slave core is defined via DPMBOOT register.

4.3 Interrupt vector tables

Each core has its own interrupt vector table and interrupt vector handler. They must be located on different locations of Flash memory.

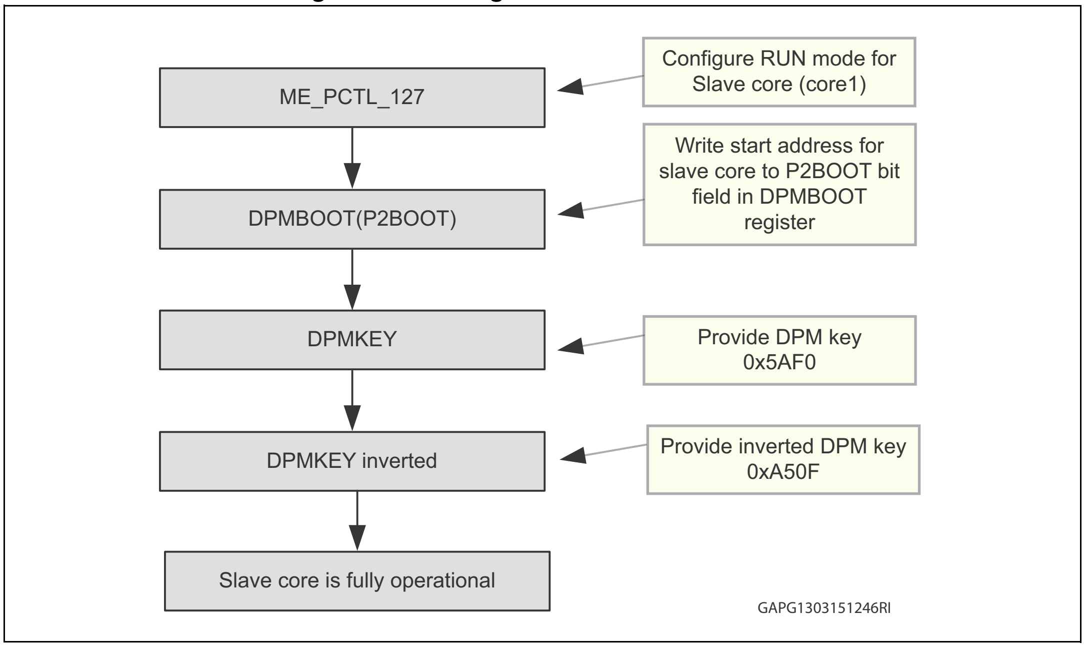
4.4 Starting of slave core (core1)

To start Slave core it is necessary to write start address of slave core into P2BOOT bit field followed with write to DPMKEY register. Below there is an example to start the slave core from function.

```
void Core1_start (void)
{
  (*(volatile int32_t *)0xC3FD8018) = 0x00008000;
  (*(volatile int32_t *)0xC3FD801C) = 0x5AF0;
  (*(volatile int32_t *)0xC3FD801C) = 0xA50F;
}
```

SPC56ELxx microcontroller has all time clocked slave core. Therefore it is not necessary to do any other operations to enable clock for slave core. Some microcontrollers don’t have clocked slave core. For example on SPC56APxx microcontroller slave core is acting like peripheral. (It is not clocked until peripheral control register for slave core (core1) is not configured).

Figure 8. Starting SPC56ELxx slave core


ME_PCTL_127
Configure RUN mode for Slave core (core1)
DPMBOOT(P2BOOT)
Write start address for slave core to P2BOOT bit field in DPMBOOT register
DPMKEY
Provide DPM key 0x5AF0
DPMKEY inverted
Provide inverted DPM key 0xA50F
Slave core is fully operational
GAPG1303151246RI

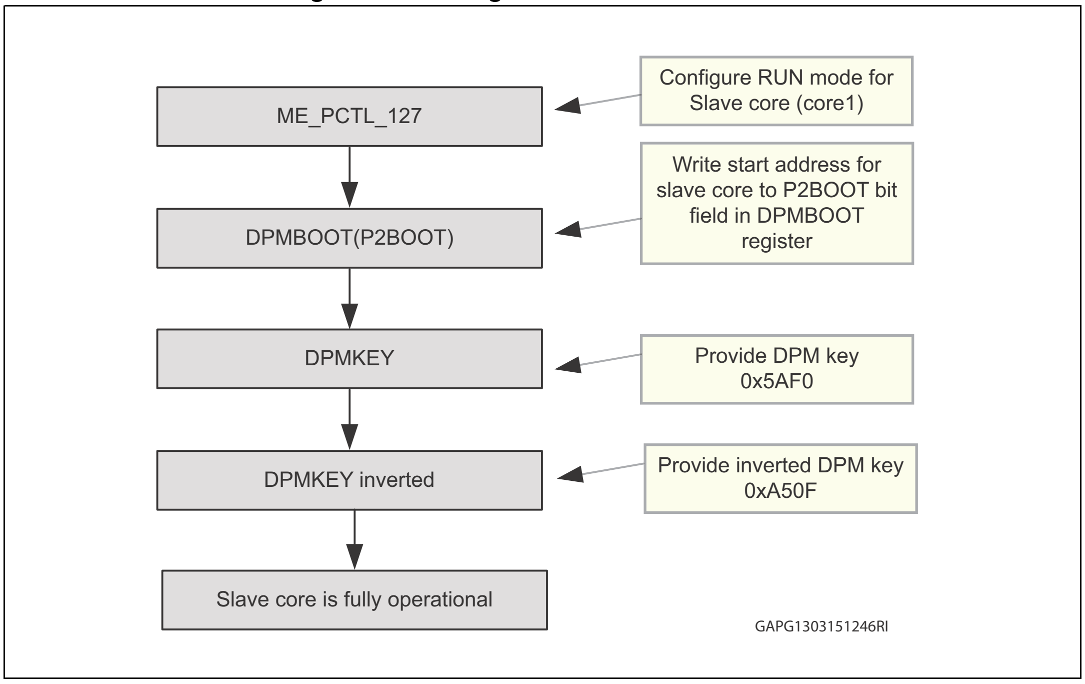
4.4.1 Starting second core of SPC56APxx microcontroller

SPC56APxx microcontroller is different than SPC56ELxx and SPC56Exx and its slave core (core1) is acting like a peripheral. This means that slave core is not clocked after reset. First thing before start writing a boot address is to provide a clock for slave core via Mode Entry

Peripheral Control Register (ME_PCTL 127). ME_PCTL_x registers select the configurations during run and non-run modes for each peripheral. Here it is possible to configure RUN, Low Power, Debug mode control slave core clock source.

In *Figure 8* it is explained the slave core starting procedure.

Figure 9. Starting SPC56APxx slave core


ME_PCTL_127
DPMBOOT(P2BOOT)
DPMKEY
DPMKEY inverted
Slave core is fully operational
Configure RUN mode for Slave core (core1)
Write start address for slave core to P2BOOT bit field in DPMBOOT register
Provide DPM key 0x5AF0
Provide inverted DPM key 0xA50F
GAPG1303151246RI

Here below it is an example function for starting slave core (core1) on SPC56APxx microcontroller:

Example 2

```
ME.PCTL[127].B.RUN_CFG = 0;  //PC0 for slave core (core1)

void Core1_start (void)
{
  DPMBOOT.R = 0x00008000;
  DPMKEY.R = 0x5AF0;
  DPMKEY.R = 0xA50F;
}
```

4.5 XBAR configuration

The purpose of the XBAR is to concurrently support up to eight simultaneous connections between master ports and slave ports. The XBAR supports a 32-bit address bus width and a 64-bit data bus width at all master and slave ports.

Multicore microcontrollers offer possibility to configure XBAR (Crossbar) bus. When a master makes an access to the XBAR the access it is immediately taken from the XBAR. If the targeted slave port of the access is available then the access is immediately presented on the slave port.

4.5.1 Example of configuration

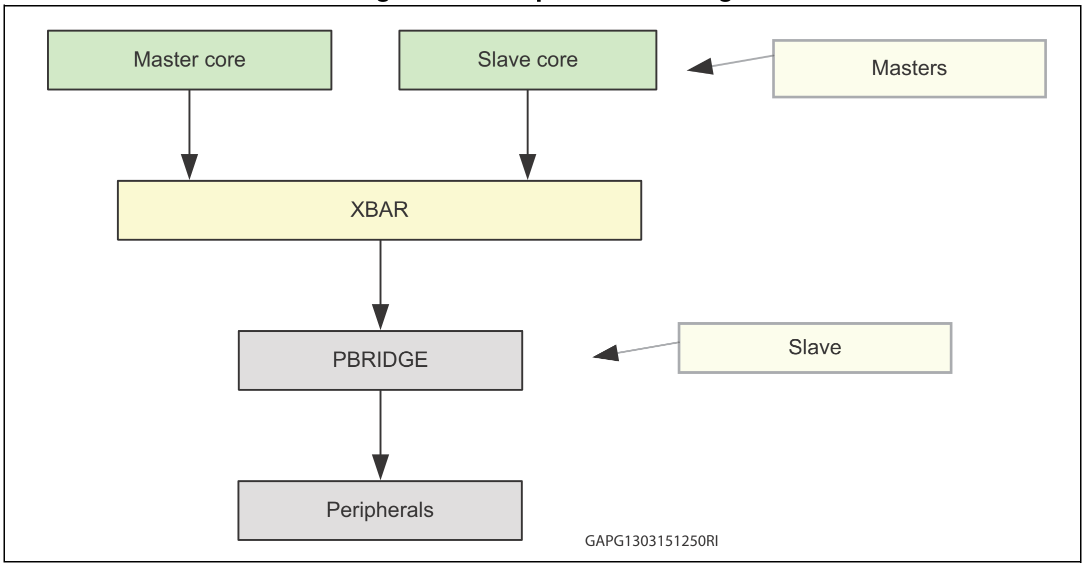
The XBAR masters are usually configured as following, when DMA or FlexRay is not occupying XBAR for long time with big data transfers:

1. DMA_0
2. FlexRay
3. Core_0 Instruction port
4. Core_0 Load/Store port
5. Core_1 Instruction port
6. Core_1 Load/Store port

4.5.2 Peripheral sharing

Figure 9 is illustrating the cores connection to the peripherals. Each core is connected to XBAR bus. Peripherals are connected to the XBAR bus through the PBRIDGE (peripheral bridge). PBRIDGE is a slave on XBAR. This means, that configuration of Mater access rights on XBAR directly affecting the PBRIDGE.

Figure 10. Peripherals sharing



4.6 Output files

Because the presented example is using 2 linker files, there are 2 output files after compilation. There is also the possibility to have one output file, but this requires 1 linker file for both cores (applications). This is commonly used in multicore operating systems. But in automotive are used, most of time, 2 separated programs for 2 cores. This is due to safety reasons.

5 Summary

This document gives a short overview of different multicore architectures developed by ST. In addition some hints to get started with multicore are provided.

Appendix A Master core MMU initialization example

```
e_lis r5, 0x10000000@ha
e_add16i r5, r5, 0x10000000@l
     mtspr mas0,r5            ; mtspr MAS0,r5
e_lis r5, 0xC0000580@ha
e_add16i r5, r5, 0xC0000580@l
      mtspr mas1,r5; mtspr MAS1,r5
e_lis r5, 0x00000020@ha
e_add16i r5, r5, 0x00000020@l
mtspr mas2,r5            ; mtspr MAS2,r5
e_lis r5, 0x0000003F@ha
e_add16i r5, r5, 0x0000003F@l
mtspr mas3,r5            ; mtspr MAS3,r5
  tlbwe                    ; Write the entry to the TLB

                ;table 1
e_lis r5, 0x10010000@ha
e_add16i r5, r5, 0x10010000@l
mtspr mas0,r5            ; mtspr MAS0,r5
e_lis r5, 0xC0000400@ha
e_add16i r5, r5, 0xC0000400@l
mtspr mas1,r5            ; mtspr MAS1,r5
e_lis r5, 0x40000020@ha
e_add16i r5, r5, 0x40000020@l

mtspr mas2,r5            ; mtspr MAS2,r5
e_lis r5, 0x4000003F@ha
e_add16i r5, r5, 0x4000003F@l
mtspr mas3,r5            ; mtspr MAS3,r5
  tlbwe                    ; Write the entry to the TLB

                ;table 2
e_lis r5, 0x10020000@ha
e_add16i r5, r5, 0x10020000@l
mtspr mas0,r5            ; mtspr MAS0,r5
e_lis r5, 0xC0000500@ha
e_add16i r5, r5, 0xC0000500@l
mtspr mas1,r5            ; mtspr MAS1,r5
e_lis r5, 0xFFF0000A@ha
e_add16i r5, r5, 0xFFF0000A@l
mtspr mas2,r5            ; mtspr MAS2,r5
e_lis r5, 0xFFF0003F@ha
e_add16i r5, r5, 0xFFF0003F@l
```

```
mtspr mas3,r5           ; mtspr MAS3,r5
  tlbwe                   ; Write the entry to the TLB

                ;table 3
e_lis r5, 0x10030000@ha
e_add16i r5, r5, 0x10030000@l
mtspr mas0,r5            ; mtspr MAS0,r5
e_lis r5, 0xC0000500@ha
e_add16i r5, r5, 0xC0000500@l
mtspr mas1,r5            ; mtspr MAS1,r5
e_lis r5, 0xFFE0000A@ha
e_add16i r5, r5, 0xFFE0000A@l
mtspr mas2,r5            ; mtspr MAS2,r5
e_lis r5, 0xFFE0003F@ha
e_add16i r5, r5, 0xFFE0003F@l
mtspr mas3,r5            ; mtspr MAS3,r5
  tlbwe                    ; Write the entry to the TLB

                ;table 4
e_lis r5, 0x10040000@ha
e_add16i r5, r5, 0x10040000@l
mtspr mas0,r5            ; mtspr MAS0,r5
e_lis r5, 0xC0000500@ha
e_add16i r5, r5, 0xC0000500@l
mtspr mas1,r5            ; mtspr MAS1,r5
e_lis r5, 0x8FF0000A@ha
e_add16i r5, r5, 0x8FF0000A@l
mtspr mas2,r5            ; mtspr MAS2,r5
e_lis r5, 0x8FF0003F@ha
e_add16i r5, r5, 0x8FF0003F@l
mtspr mas3,r5            ; mtspr MAS3,r5
  tlbwe                    ; Write the entry to the TLB

                ;table 5
e_lis r5, 0x10050000@ha
e_add16i r5, r5, 0x10050000@l
mtspr mas0,r5            ; mtspr MAS0,r5
e_lis r5, 0xC0000500@ha
e_add16i r5, r5, 0xC0000500@l
mtspr mas1,r5            ; mtspr MAS1,r5
e_lis r5, 0xC3F0000A@ha
e_add16i r5, r5, 0xC3F0000A@l
mtspr mas2,r5            ; mtspr MAS2,r5
e_lis r5, 0xC3F0003F@ha
e_add16i r5, r5, 0xC3F0003F@l
```

```
mtspr mas3,r5            ; mtspr MAS3,r5
  tlbwe                    ; Write the entry to the TLB

    se_isync
```

Revision history

Table 2. Document revision history

Date	Revision	Changes
04-Dec-2015	1	Initial release.