life.augmented

# AN4389
# Application note

## SPC574K72/SPC57EM80 Getting Started

## Introduction

The SPC574K72/SPC57EM80 devices family is built on Power Architecture® technology and is targeted for automotive powertrain controller applications, chassis control applications, transmission control applications, steering and braking applications, as well as low-end hybrid applications and safety applications that require a high safety integrity level (ISO 26262 for ASIL-D safety integrity).
In order to minimize some additional software and module level features to reach this target, an on-chip redundancy is offered for the critical components of the microcontroller (see Functional Safety chapter of RM) by multiple CPU computational cores with delayed lockstep, an I/O processor core, a DMA controller, an interrupt controller, a dual crossbar bus system, a memory protection unit, a fault collection and control unit (FCCU), flash an end-to-end error correction coding (ECC).
This family operates up to 200MHz and offers a high performance processing optimized moreover if compared with the previous family (SPC56) within a similar power envelope. Some hardware in the new family also helps to prevent and control critical electronics system faults and protects against harmful hacks.
This application note details the steps required to properly initialize the SPC574K72/SPC57EM80 devices family from power-up to the code execution as well as how to control the boot-up of the available cores. An example code is described throughout the application note to explain the steps.

It is intended that this application note is read along with the SPC574K72/SPC57EM80 Reference Manuals, that can be obtained from the STMicroelectronics website at http://ww.st.com (see *Appendix E: Further Information*).

# Contents


1 **Application example description** . . . . . 6
 1.1 Limitations . . . . . 7

2 **Family architecture** . . . . . 8
 2.1 I/O Processor . . . . . 9

3 **Reset and boot** . . . . . 10
 3.1 Modules involved in the Reset and Boot sequence . . . . . 11
  3.1.1 Power Management Controller . . . . . 11
  3.1.2 Reset Generation Module . . . . . 11
  3.1.3 SSCM . . . . . 11
  3.1.4 BAF . . . . . 14
  3.1.5 Boot Header format . . . . . 16
  3.1.6 Mode Entry Module . . . . . 17
 3.2 Boot from internal flash . . . . . 17
  3.2.1 From power-up to code execution . . . . . 17

4 **SPC574K72/SPC57EM80 initialization example** . . . . . 19
 4.1 A valid Boot Header . . . . . 20
 4.2 Software Watchdog handling . . . . . 21
 4.3 Core registers initialization . . . . . 22
  4.3.1 EABI Register initialization . . . . . 22
  4.3.2 Core 0 Register initialization . . . . . 24
 4.4 Enable Branch Target Buffer . . . . . 24
 4.5 SRAM ECC Initialization . . . . . 24
 4.6 Environment Initializations . . . . . 27
 4.7 XBAR Configuration . . . . . 28
  4.7.1 Flash Memory Access . . . . . 30
  4.7.2 XBAR Register configuration . . . . . 31
 4.8 Memory Controllers configuration . . . . . 32
  4.8.1 Flash Controller configuration . . . . . 32
  4.8.2 SRAM wait State . . . . . 33
 4.9 Mode Entry Module: Configuration . . . . . 33

4.9.1 Core Control Register Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.10 Clock and PLL configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.11 Cache configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**5 Blink LED application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40**

**Appendix A Copy Initialized Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41**

**Appendix B Cache Configuration source code . . . . . . . . . . . . . . . . . . . . . . . . . 43**

**Appendix C Core0 registers initialization code . . . . . . . . . . . . . . . . . . . . . . . . 46**

**Appendix D Clocks and PLLs Initialization example. . . . . . . . . . . . . . . . . . . . . 48**

**Appendix E Further Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50**
E.1 Reference documents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
E.2 Acronyms and abbreviations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

**Revision history . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52**

# List of tables

Table 1. Boot Header Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
Table 2. Boot CPU Selection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
Table 3. Power Architecture EABI Registers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
Table 4. Flash memory port assignment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
Table 5. Core Control Register x (ME_CCTLx) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
Table 6. Core Address Register x (ME_CADDRx) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
Table 7. Acronyms and abbreviations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
Table 8. Revision history . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

# List of figures


Figure 1. Project handling: a unique environment for all the cores . . . . . . 6
Figure 2. Project handling: an independent application for each core . . . . . . 7
Figure 3. SPC574K72 Block diagram . . . . . . 8
Figure 4. SPC57EM80 Block diagram . . . . . . 9
Figure 5. Modules involved in Reset and Boot process . . . . . . 10
Figure 6. SSCM: Boot-up sequence . . . . . . 13
Figure 7. BAF Data Flow Diagram . . . . . . 15
Figure 8. Core Execution flow . . . . . . 19
Figure 9. SPC574K72/SPC57EM80 memory and cores allocation . . . . . . 25
Figure 10. SRAM ECC Initialization flow . . . . . . 26
Figure 11. Booting flow using GHS startup libraries . . . . . . 28
Figure 12. Crossbar Switch diagram . . . . . . 29
Figure 13. Computational Shell’s crossbar switch integration . . . . . . 30
Figure 14. Peripheral Shell’s crossbar switch integration . . . . . . 31
Figure 15. 2-way, 4-entry mini-cache organization . . . . . . 32
Figure 16. Mode Entry Diagram . . . . . . 34
Figure 17. SPC574K72/SPC57EM80 Clock Generation . . . . . . 37
Figure 18. Cache configuration flow . . . . . . 39
Figure 19. Blink LED application . . . . . . 40

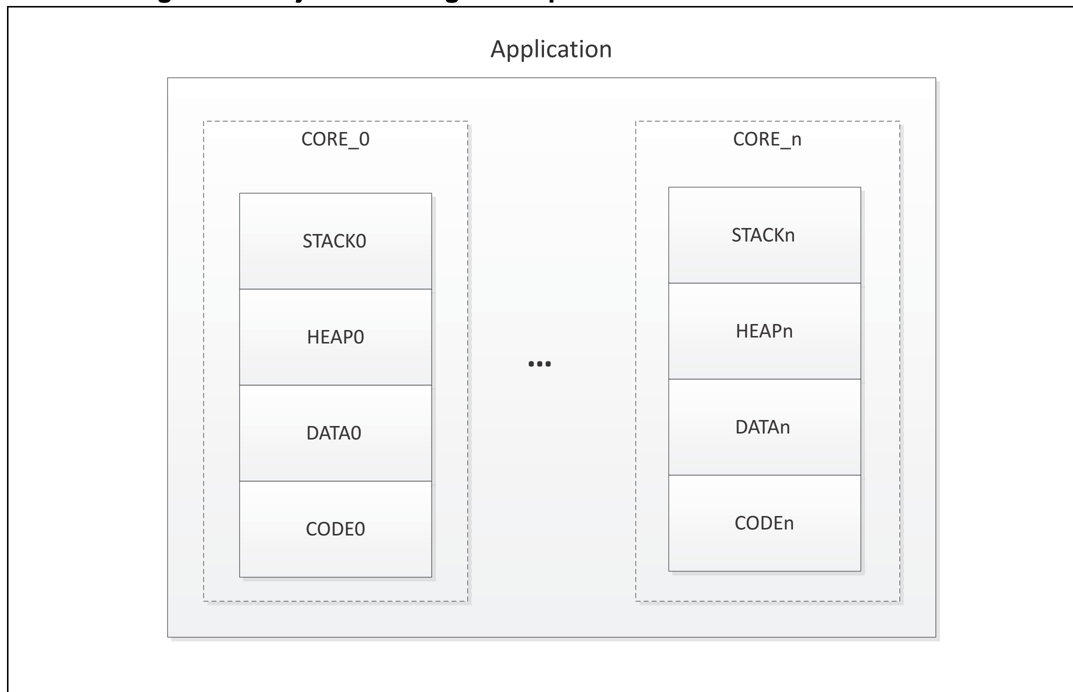
# 1 Application example description

This application note describes the necessary steps to configure the device in order to boot-up from the Flash and run independent codes on each cores of the SPC574K72/SPC57EM80 devices family.
This family is quite different from the previous one (SPC56). It is built on 55nm technology and indeed apart from the technological point of view (55nm versus the older 90nm).
It has been designed to help the user to obtain an ISO26262 ASIL-D compliance for his applications.
With respect to the old technology, this multicore architecture allows the user to have a high computation power with a low power consumption.

The example code described herein toggles one LED for each core and is one application for all cores even if it has been chosen to have a unique environment shared between the cores (see *Figure 1*).
This choice implies that the user has to take care in the core initializations, the data copy from the Flash to RAM and moreover in stack/ram allocation for each core.

The user can choose to use the compiler support to do these initializations for one core[a], while for the others have to provide the initialization steps.

**Figure 1. Project handling: a unique environment for all the cores**

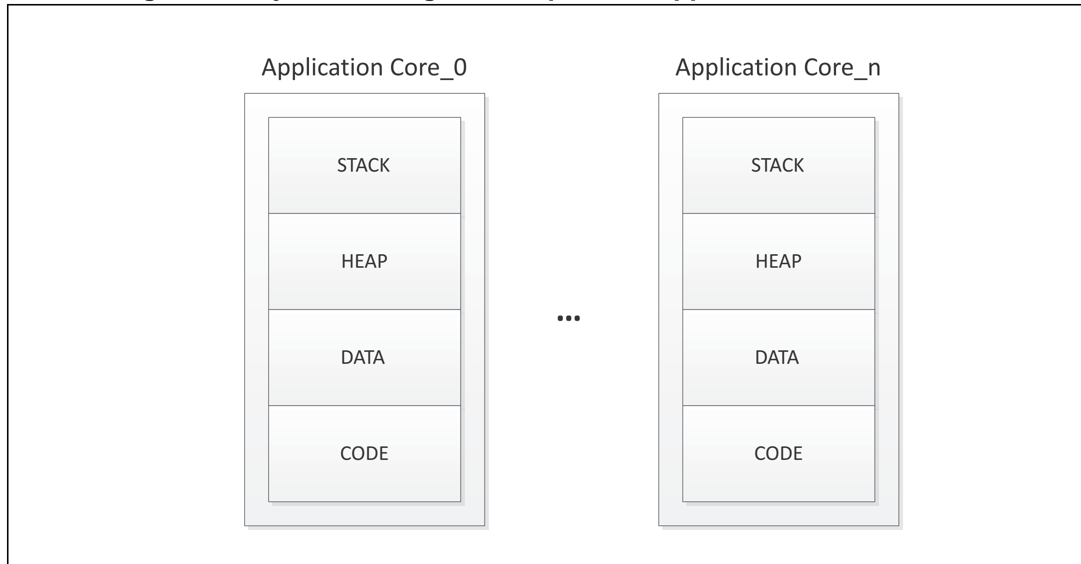



The compiler support (through the compiler property libraries) (see *Figure 11: Booting flow using GHS startup libraries*) allows the user to avoid some core register initializations (see *Section 4.3.2: Core 0 Register initialization* for the exceptions) as well as the data copy of initialized data from the Flash to SRAM (see *Section 4.6: Environment Initializations*) before

a. The GHS compiler supports only one application (instance).

the user code execution starts.
From execution point of view each core has its own execution flow while from the compiler point of view it has a unique image where the data and the code of all the cores cohabit.
Of course the user can choose to have an independent application for each core (see *Figure 2: Project handling: an independent application for each core*) with or without the compiler support for the environment initializations (see *Figure 11: Booting flow using GHS startup libraries*).

**Figure 2.Project handling: an independent application for each core**




*Note: Even if there are no limitations on the use of whatever compiler/toolchain, in this document the GHS (Green Hills) syntax has been used where the code has been inserted as example.*

## 1.1 Limitations

Since the scope of this document is an introduction on how to start to work with this devices family, the initialization flow described doesn't include any safety/security topics as well as doesn't include any special memory handling like internal memory remapping or memory protection levels (see *E.1: Reference documents*).

# 2 Family architecture

The device architecture is split into two distinct regions, the computational shell and the I/O subsystem or peripheral shell (see *Figure 3: SPC574K72 Block diagram* and *Figure 4: SPC57EM80 Block diagram*).
The main computational shell consists of e200z4420 CPU with an identical CPU running as a safety checker core in a delayed lockstep mode (the SPC57EM80 has another identical CPU used for performance purposes) and a fast system memory connected through a fast crossbar switch (XBAR_0).
The I/O subsystem consists of an I/O processor (IOP) (see *Section 2.1: I/O Processor*) targeted to manage the peripherals. The SPC57EM80 has an e200z425 while the SPC574K72 has an e200z225.

*Note: The SPC57EM80 has an additional programmable CPU (e200z0h) embedded in the Hardware Security Module (HSM).*

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

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.

**Figure 3.SPC574K72 Block diagram**

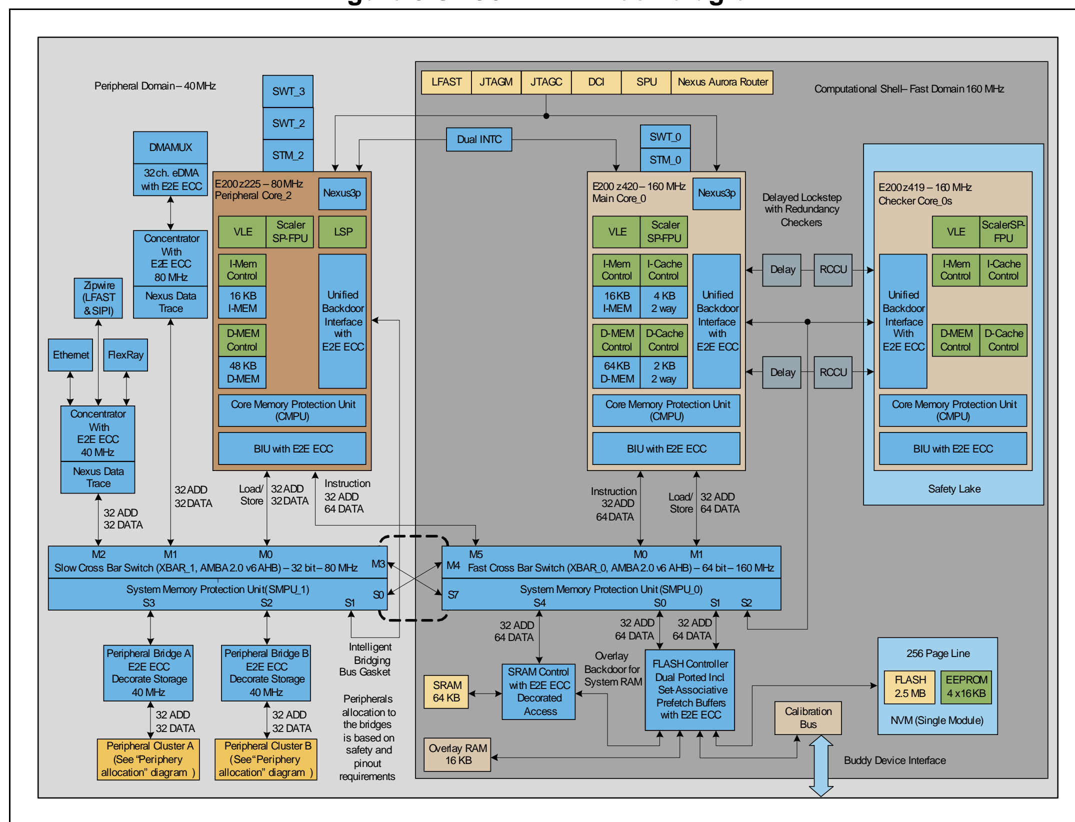
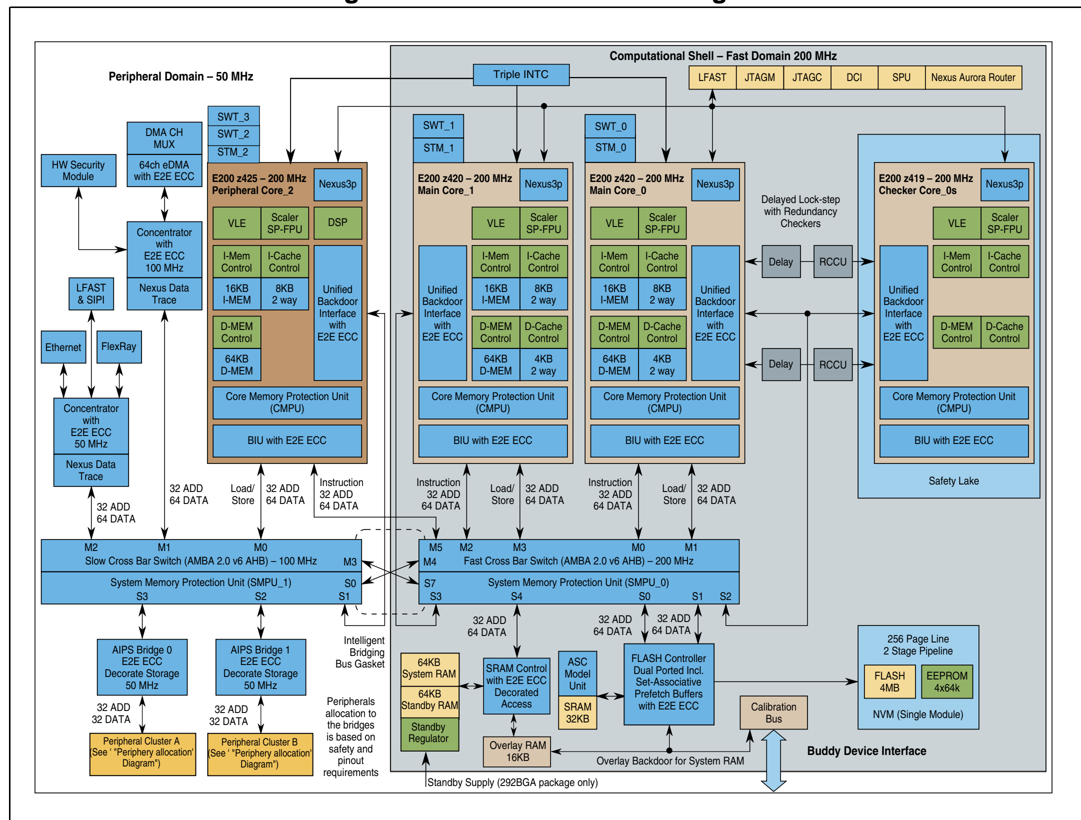


**Figure 4.SPC57EM80 Block diagram**




## 2.1 I/O Processor

The I/O processor (IOP), or Peripheral Core_2, is one of the processor cores on the SPC574K72/SPC57EM80 family devices (see *E.1: Reference documents*).

The IOP is intended to configure the machine and control various device peripheral elements.

The IOP is automatically started during the boot-up sequence after the release of RESET (see *Section 3.2: Boot from internal flash*).

During the boot sequence, the System Status and Configuration Module (SSCM) reads the device configuration records (DCF) for the IOP start address in the Boot Assist Flash (BAF) code (see *Section 3.1: Modules involved in the Reset and Boot sequence*).

This information is stored in the Mode Entry Module (ME) until the internal reset signal is released and then the IOP can begin code execution at the designated location.

SPC574K72/SPC57EM80 devices are pre-configured to begin the IOP code execution at a start address located in the BAF (see *Section 3.1.4: BAF*) with a default start-up code.

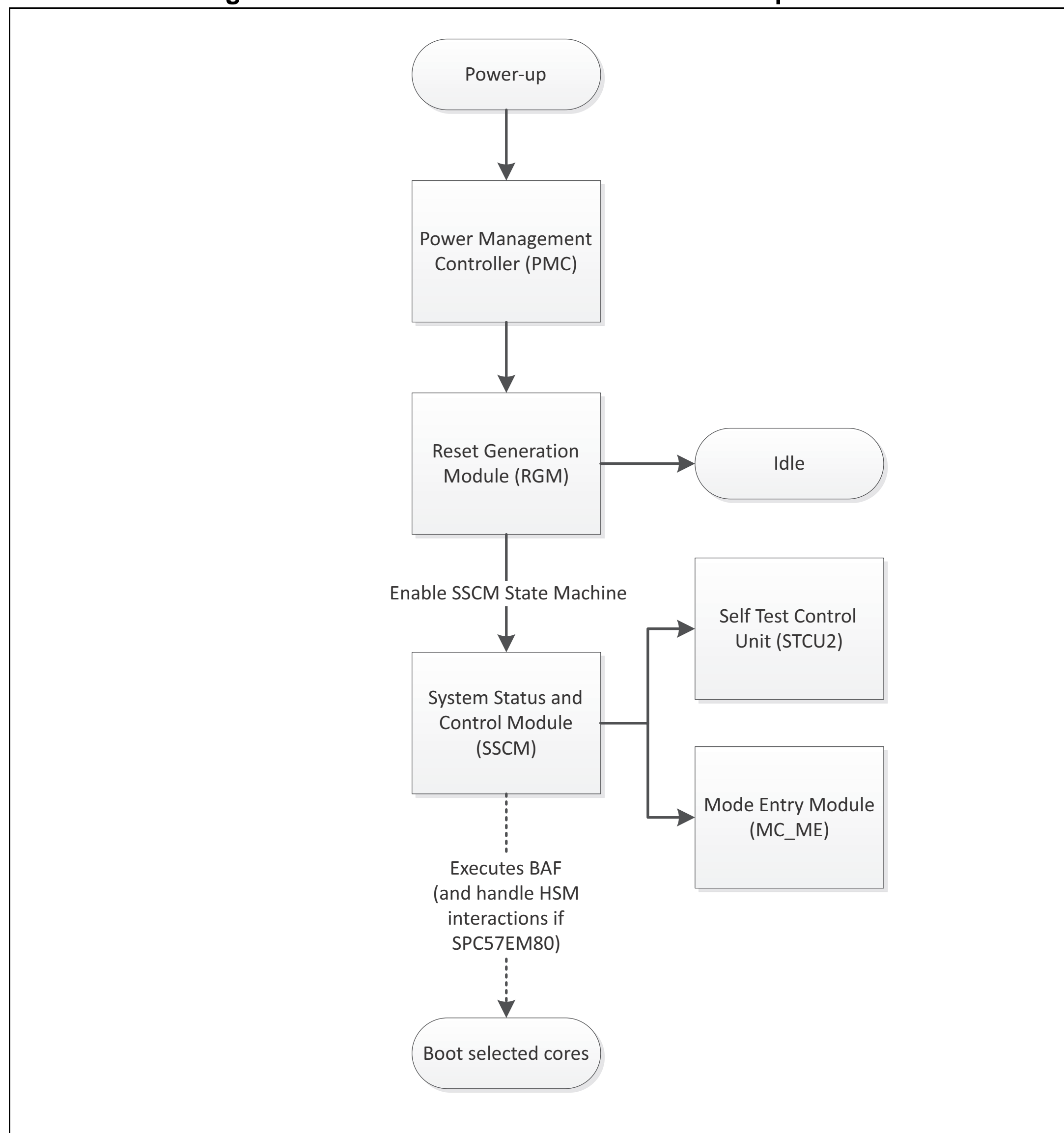
In the absence of a user application code, the BAF initiates download of any user code via the LINFlexD port and the program execution begins at the start address embedded in the downloaded code.

Once the IOP has performed the initialization via a preloaded instruction code in the BAF and downloaded any eventual application code, the IOP can be programmed to perform any duty or access to any memory element.

# 3 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 SPC574K72/SPC57EM80 family devices and various modes enabled from settings in the DCF records.
There are several modules used to step through the sequence needed to properly reset the SPC574K72/SPC57EM80 and prepare it to fetch the first instruction of the user application code.
These are the Power Management Controller (PMC), the Reset Generation Module (RGM), the Mode Entry Module (MC_ME), the System Status and Configuration Module (SSCM) and the Self-Test Control Unit (STCU2) (see *Figure 5: Modules involved in Reset and Boot process*).

**Figure 5. Modules involved in Reset and Boot process**



The complex boot allows the user to control several configuration parameters and customize the final behavior of the device (see *E.1: Reference documents*).
For instance, the SPC574K72/SPC57EM80 family devices enter in a serial boot mode if a valid boot header file is not found. In this case, the device uses the serial boot loader to receive the startup code and begin the program execution. If an emulation and debug device is detected during the reset sequence, the device enters in the calibration sequence[b].
Another reset outcome is that the Hardware Security Module (only for the SPC57EM80), the IO Processor, and the boot CPU are all properly started and begin executing their respective program code.
Since this document is focused on how to boot-up the device from the internal flash (with no use of security functions[c]) in the following sections will be detailed how the user has to configure the device (and its own code) to choose the proper behavior and the core/s from which the device boots.

## 3.1 Modules involved in the Reset and Boot sequence

### 3.1.1 Power Management Controller

The Power Management Controller (PMC) controls and monitors various voltage levels around the chip:

- its own supply voltage
- the supply voltages to all the high- and low-voltage detect circuits
- the trip points for all the high- and low-voltage detect circuits
- the power supplies and reference voltages to the Analog-to-Digital Converter
- the major power supplies to the SPC574K72/SPC57EM80 family devices

### 3.1.2 Reset Generation Module

The Reset Generation Module (RGM) is a complex state machine that begins sequencing the SPC574K72/SPC57EM80 family devices through the initial part of the reset process. Reset sources are organized into two categories: destructive and functional.
The RGM is simply a state machine. It does not execute program code.
In general, the reset generation module centralizes the different reset sources and manages the reset sequence.

### 3.1.3 SSCM

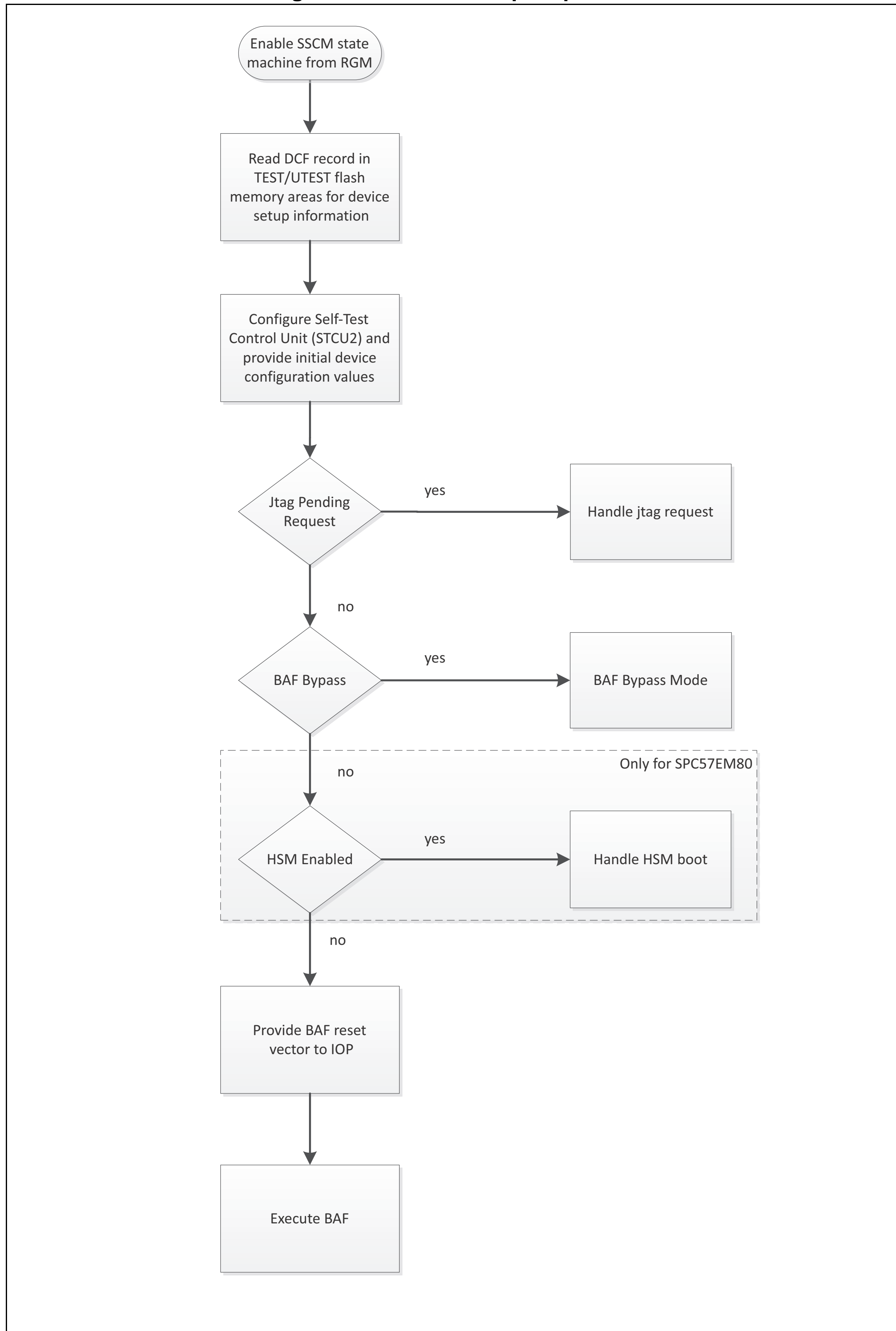
The System Status and Configuration Module (SSCM) is a state machine. During the reset sequence, the Reset Generation Module (RGM) enables the SSCM which continues with the reset or boot-up sequence (see *Figure 6: SSCM: Boot-up sequence*). Once the SSCM locates the boot header (see *Table 1: Boot Header Structure*), the SSCM writes the start

b. The calibration sequence is not a user mode and is only used for code development.

c. There aren’t explained the steps necessary to run (for SPC57EM80) code on HSM.

address information for the CPUs (cores) to special locations in the Mode Entry Module (MC_ME) (see *Section 4.9: Mode Entry Module: Configuration*).
For all cores enabled by the boot header, the MC_ME feeds a reset vector to each enabled core and the respective cores begin the program execution at the specified address. The SSCM starts up the IOP and the HSM (only for the SPC57EM80) in different ways depending on whether there is valid code in the Boot Assist Flash (BAF).
The boot-up sequence ends when the SSCM starts up the boot CPU, which can be the Core_2 (IOP), the Core_0 or the Core_1 (only for the SPC57EM80).

**Figure 6.SSCM: Boot-up sequence**



### 3.1.4 BAF

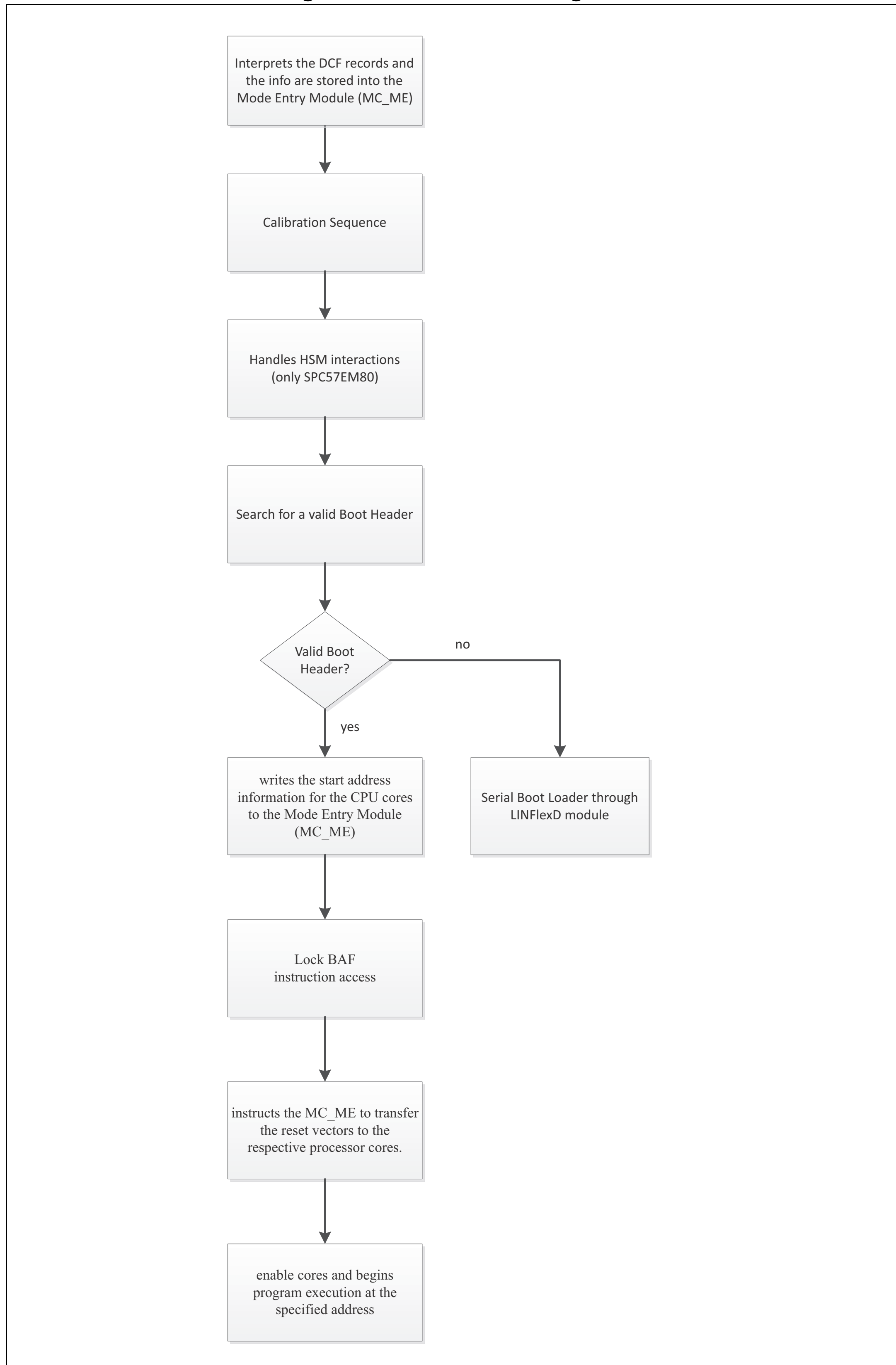
The Boot Assist Flash (BAF) contains some code written by the manufacturer (see *Figure 7: BAF Data Flow Diagram*):

- it facilitates the boot-up procedure of the SPC574K72/SPC57EM80 family devices
- it searches for a valid boot header and boot the device from the internal flash
- it starts and runs the Serial Boot Loader (if no valid boot header is found)

*Note: If there is no valid boot header loaded into the program memory when the SPC574K72/SPC57EM80 devices family are powered on, the Serial Boot Loader software can receive files from an external serial link controlled by the internal LINFlexD module. The received file is put into RAM and then the program execution begins at the start address specified in the received file.*

The serial boot loader is only an option in the early stages of the device life cycle.
In later life cycle stages (for example “field”) the device simply waits for the watchdog to reset the device if no boot header is found.

**Figure 7.BAF Data Flow Diagram**



### 3.1.5 Boot Header format

The boot header is used to supply the start address for some code execution for the Core_0, the Core_1(only for SPC57EM80) and the Checker Core_0s. The boot header is written by the user and loaded into specific locations in the flash memory (bootable sectors) (see *Section 3.2.1: From power-up to code execution*). The boot header contains the reset vectors for the various cores and it also specifies the BOOT CPU[d] that eventually execute application code.

**Table 1. Boot Header Structure**

| Offset | Field | | Description |
|---|---|---|---|
| 0x0 | Boot Header_ID | Boot_CPU | Valid Boot header identifier is 0x005A \| Boot CPU |
| 0x4 | Start Address | | Reset Vector for I/O Processor (Core_2) |
| 0x8 | Configuration Bits | | Default is 0x0000_0000. Reserved for Code Size field for CSE based devices |
| 0xC | Configuration Bits | | Default is 0x0000_0000. Reserved for future use |
| 0x10 | CPU_0 Reset Vector | | Reset Vector for Core_0 |
| 0x14 | CPU_1 Reset Vector | | Reset Vector for Core_1[1] |
| 0x18 | CPU_0SC Reset Vector | | Reset Vector for Core_0 Checker |

1. Valid only for SPC57EM80 (0xFFFFFFFF for SPC574K72)

The Boot_CPU is read by the host during the CPU (Core_2) is executing the BAF, and used to select the CPU to execute the user application code. Mode Entry Module contains the register interface to configure the reset vector for each CPU and apply the mode change to select the CPU.

d. It's possible to boot up one or more core simultaneously: in the example described in this document the focus is on starting from the Core_2 that will later triggers the start of the other(s).

Table 2. Boot CPU[1] Selection

| Boot_Header_ID bit | CPU Selected |
|---|---|
| Boot_Header_ID[0] | I/O Processor (Core_2) |
| Boot_Header_ID[1] | Core_0 |
| Boot_Header_ID[2] | Core_0SC |
| Boot_Header_ID[3][2] | Core_1 |
| Boot_Header_ID[4-15] | Reserved for future use |

1. Only four bits in the 16-bit Boot_CPU field are used.
2. Valid only for SPC57EM80

*Note: Setting of Boot_Header_ID[2] is only valid if Boot_Header_ID[1] is also set.*

### 3.1.6 Mode Entry Module

The Mode Entry module (MC_ME) is responsible for delivering initial values to many registers in SPC574K72/SPC57EM80 family devices. Among other things, the MC_ME supplies the reset vectors to all the cores.
During the power-up sequence, the System Status and Configuration Module (SSCM) searches for a boot header at defined locations in the flash memory (see *E.1: Reference documents*), derives the necessary reset vectors from the boot header and then writes these vectors to their respective locations in the Mode Entry module. The SSCM also interprets the DCF records, which contain other information that is written to the MC_ME. As the boot-up sequence progresses, the SSCM instructs the MC_ME to transfer the reset vectors to the respective processor cores.

Even though the reset vectors are in the boot header located in the flash memory, the Mode Entry module always provides them to the CPU cores.

## 3.2 Boot from internal flash

In this scenario, a Boot CPU is selected and starts executing some application code. Also, this is the scenario that is used to start all processor cores except the HSM (only for the SPC57EM80): the Core_0, Core_1(only for the SPC57EM80), the Checker Core_0s, and the IOP (Core_2).

### 3.2.1 From power-up to code execution

The initial power-up sequence starts with the power being applied to the SPC574K72/SPC57EM80 devices family.
The Reset Generation Module (RGM) then advances the device through a series of steps (reset sequence): starts the System Status and Control Module (SSCM), which continues the boot-up process (it takes control) after the RGM enters the IDLE state (see *Figure 5: Modules involved in Reset and Boot process*).
At this point, LBIST and MBIST have completed their test cycles, the data in the DCF record has been written to the appropriate registers and all the trim values for the analog portions of the chip have been installed in their proper locations.

In the hypothesis of booting from the internal flash (see also Reset and Boot Chapter of document listed into *E.1: Reference documents*) the SSCM releases reset to the I/O Processor (IOP)[e] and then turns on the clock to the IOP.
The BAF code start address is transferred to the IOP at this time under the control of the SSCM and the execution of the Boot Assist Flash code begins[f].
The BAF code begins by initializing the registers and applying the device settings.

*Note: The BAF code, written by the device manufacturer, changes from SPC574K72 to SPC57EM80.*

At this point if an emulation device is not detected and the HSM is not enabled the IOP continues the code execution by searching for the boot header. In the hypothesis of this document, it is assumed that a valid Boot Header is present.
This is the header file that specifies which Boot CPU starts executing the user application code.
The Boot CPU is either the IOP, the Core_0 or the Core_1 (only for SPC57EM80). The boot header file also contains the reset vector addresses for the Core_0, the Core_1 (only for SPC57EM80), the Core_2 (IOP) and the Checker Core_0s.
The Boot CPU header is programmed into the flash memory at the same time the user application program is programmed into flash memory.
The IOP searches for the boot header, which can be located at one of the eight addresses:

- 0x00FC_0000
- 0x00FC_4000
- 0x00FC_8000
- 0x00FC_C000
- 0x0100_0000
- 0x0104_0000
- 0x0108_0000
- 0x010C_0000

A valid boot header (Boot Header_ID begins with the value 0x005A) is assumed to be present in the flash memory. In this case the I/O Processor decodes the boot header and then writes the reset vectors for the Core_0, the Core_1 (only for the SPC57EM80), the Core_2 and the Checker Core_0s to the Mode Entry Module (MC_ME).

The last steps before the boot cores execution are:

- Access locking to the Boot Assist Flash
- Transfer the reset vectors from MC_ME to the respective CPUs and the code execution, for each of the cores, begins at their respective reset vector.

e. During PHASE3[DEST] of the Reset Generation Module's sequence, the start address for execution of the BAF code loaded into the Boot Assist Flash is transferred to the MC_ME.

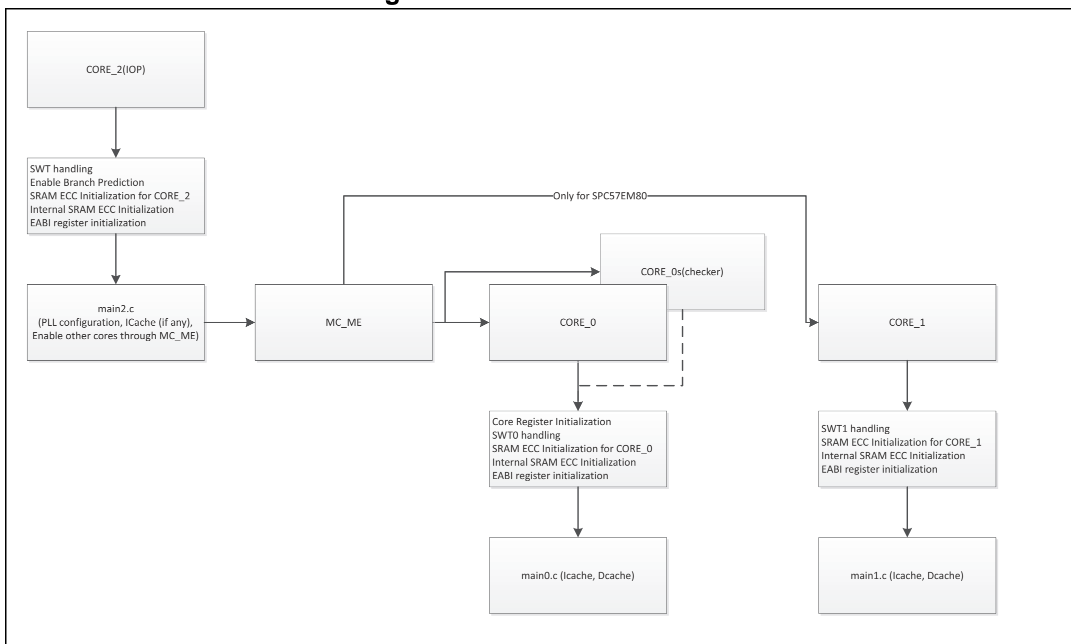
f. The IOP executes BAF code, written by the manufacturer, to apply specific device settings to the SPC574K72/SPC57EM80 family devices.

# 4 SPC574K72/SPC57EM80 initialization example

As stated in the previous chapters in this document it is described how the device boot-up from the IOP (see *Section 3.2: Boot from internal flash* and *Section 3.1.5: Boot Header format*).
*Figure 8* shows the core execution flow followed in this document.

As soon the device initialization is done (general architecture initialization[g]) the other cores (only Core_0 for the SPC574K72) are enabled and started through the Mode Entry Module (MC_ME). This operation in any case implies a minimum set of initialization steps indeed each core has its own environment (Caches, BTB, TCM, core registers).

**Figure 8.Core Execution flow**




g. There are no special rules about the timeline to follow in configuration flow: the only exception is the timeout related to SWT2 that user has to handle.

In order to run an application from the Flash memory, the user code have to perform a series of actions for a proper execution:

- Provide a valid boot header (see *Section 4.1: A valid Boot Header*)
- Initialize the Core Registers (see *Section 4.3: Core registers initialization*)
- Handle/configure the SWT (Software Watchdog) (see *Section 4.2: Software Watchdog handling*)
- Enable the BTB (Branch Target Buffer) (see *Section 4.4: Enable Branch Target Buffer*)
- Initialize the SRAM (ECC) (see *Section 4.5: SRAM ECC Initialization*)
- Configure the XBAR (see *Section 4.7: XBAR Configuration*)
- Configure the wait states for Flash and SRAM (see *Section 4.8: Memory Controllers configuration*)
- Configure the Mode (see *Section 4.9: Mode Entry Module: Configuration*)
- Configure the PLL and clock selectors/dividers (see *Section 4.10: Clock and PLL configuration*)
- Configure the Caches (see *Section 4.11: Cache configuration*)
- Enable and start the cores through Mode Entry Module (MC_ME) (see *Section 4.9.1: Core Control Register Configuration*)

*Note: The user can do these initialization either in assembly or in C code.*

## 4.1 A valid Boot Header

In order to boot up correctly the device (in the hypothesis of this document) from the IOP the user has to provide a valid boot header (see *Section 3.1.5: Boot Header format*).

Below a scratch of code to define (with GHS syntax) a valid boot header for SPC574K72:

```
.LONG 0x005A0001 ;#/* BH conf: IOP Only - Can boot others via ME */
.LONG _start;#/* IOP Reset Vector */
.LONG 0x00000000 ;#/* Configuration Bits */
.LONG 0x00000000 ;#/* Configuration Bits */
.LONG _start_core0 ;#/* CPU0 Reset Vector */
.LONG 0XFFFFFFFF ;#/* No CPU1 on SPC574K72*/
.LONG _start_core0 ;#/* LSCPU Reset Vector */
.LONG 0x00000000 ;#/* Padding */
```

while for SPC57EM80 it becomes:

```
.LONG 0x005A0001 ;#/* BH conf: IOP Only - Can boot others via ME */
.LONG _start;#/* IOP Reset Vector */
.LONG 0x00000000 ;#/* Configuration Bits */
.LONG 0x00000000 ;#/* Configuration Bits */
.LONG _start_core0 ;#/* CPU0 Reset Vector */
.LONG _start_core1 ;#/* CPU1 Reset Vector */
.LONG _start_core0 ;#/* LSCPU Reset Vector */
.LONG 0x00000000 ;#/* Padding */
```

As it’s possible to view the main difference, between the two boot header definitions, is on CPU1 reset vector initialization value (see *Table 1: Boot Header Structure* and *Table 2: Boot CPU Selection*).

The labels “_start”, “_start_core0” and “_start_core1” (only for SPC57EM80) represent respectively the start address for the Core2 (IOP), the Core0 and the Core1 (only for SPC57EM80) defined in the source code and/or the linker file.

*Note: In the above boot header definition, the Core0 and Core1 start addresses have been left as reference. Since the boot header is starting only the IOP (BOOT CPU = 0x1) there is no need to provide other reset vectors.*

*Note: The user can choose to boot-up from whatever core (also from all together) only by changing the boot CPU selection in the first raw of the boot header definition (see Section 3.1.5: Boot Header format).*
*For example change the first raw to LONG 0x005A000F implies that all the cores are booted.*

## 4.2 Software Watchdog handling

The Software Watchdog Timer (SWT) is a peripheral module that can prevent some system lockup in situations such as the software is getting trapped in a loop or if a bus transaction fails to terminate.

When enabled, the SWT requires periodic execution of a watchdog servicing operation. In order to prevent a system reset the watchdog must be serviced or disabled prior to the initial expiry of the timer.

The SPC574K72/SPC57EM80 device family has one SWT per core[h] but by default only SWT2 (related to the IOP) is enabled. The other SWT (only SWT0 for SPC574K72) have to be enabled by the user.

In a real application it is expected that the SWT would be serviced (before the timer expires) and reconfigured to match the application timing rather than being disabled.

Look at Reference Manual (see *E.1: Reference documents*) for more information on configuring and using the SWT.

In this document, the watchdog is disabled to avoid any servicing.

The user has to take care that there is enough time (versus the SWT time period) between the start of the initialization code (startup code) and the SWT handling function.

In order to disable the watchdog in the software application the user needs to:

- Write the sequence of 0xC520 followed by 0xD928 to the service register. This clears the soft lock bit enabling the next step in the process;
- Clear the WEN bit in the Control register;

h. There is also a security watchdog timer (see *E.1: Reference documents*)

Below a piece of code[i] that disables the SWT:

```
;#/*==============================================================*/
;# macro to turn off swt
;# \param[in] swt_baseaddr SWT IP Base address

.macro swt_disable swt_baseaddr

 e_li    r4, swt_baseaddr@h
 e_or2i r4, swt_baseaddr@l

 e_li    r3, 0xC520
 e_stw   r3, 0x10(r4)

 e_li    r3, 0xD928
 e_stw   r3, 0x10(r4)

 e_lis   r3, 0xFF00
 e_or2i  r3, 0x010A

 e_stw   r3, 0(r4)
.endm
```

*Note: When the user is connecting to the device using a debugger, it is likely that the debugger itself disables the watchdog to allow the debug to be carried out. In the other case if it's not done this can result in a fairly common problem when attempting to run the code in a standalone configuration where a periodic device reset is observed, caused by the SWT time-out.*

## 4.3 Core registers initialization

### 4.3.1 EABI Register initialization

The Power Architecture Enhanced Application Binary Interface (EABI) specifies certain general purpose registers as having special meaning for the C code execution. It defines the system interface for the compiled programs (see *Table 3: Power Architecture EABI Registers*).

i. A macro/.endm block defines the contents of a macro in GHS syntax

**Table 3. Power Architecture EABI Registers**

| Register | Content |
|---|---|
| GPR1 | Stack Frame Pointer |
| GPR2 | _SDA2_BASE |
| GPR13 | _SDA_BASE |
| GPR31 | Local variables or environment pointer |
| GPR0 | Volatile - may be modified during linkage |
| GPR3, GPR4 | Volatile - used for parameter passing and return values |
| GPR5-GPR10 | Used for parameter passing |
| GPR11-GPR12 | Volatile - may be modified during linkage |
| GPR14-GPR30 | Used for local variables |
| FPR0 | Volatile register |
| FPR1 | Volatile - used for parameter passing and return values |
| FPR2-FPR8 | Volatile - used for parameter passing |
| FPR9-FPR13 | Volatile registers |
| FPR14-FPR31 | Used for local variables |

The symbols _SDA_BASE and _SDA2_BASE are defined during the linking. They specify the locations of the small data areas. A program must load these values into GPR13 and GPR2, respectively, before accessing to the program data.
The small data areas contain part of the data of the executable. They hold a number of variables that can be accessed within a 16-bit signed offset of _SDA_BASE or _SDA2_BASE.
A total of 64k (plus or minus a 32 K offset) bytes may be addressed without changing the value in the base registers. The 16-bit displacement fits, along with the instruction op-code, into a single instruction word. This implies a more memory efficient method of accessing a variable than referencing it by using a full 32-bit address (one instruction to access to it instead of two). Typically, the small data areas contain program variables that are less than or equal to 8 bytes in size, although this differs by compiler. The variables in SDA2 are read-only.
Before executing some user code, the startup code must also set up the stack pointer in GPR1. This pointer must be 8-byte aligned for the EABI and should point to the lowest allocated valid stack frame. The stack grows toward lower addresses, so its location should be selected so that it does not grow into data or bss areas.

Below an example code to initialize these three pointers:

```
e_lis r1, __SP_INIT@h ;# Initialize stack pointer r1 to
e_or2i r1, __SP_INIT@l ;# value in linker command file.
e_lis r13, _SDA_BASE_@h ;# Initialize r13 to sdata base
e_or2i r13, _SDA_BASE_@l ;# (provided by linker).
e_lis r2, _SDA2_BASE_@h ;# Initialize r2 to sdata2 base
e_or2i r2, _SDA2_BASE_@l ;# (provided by linker).
```

Note: *This initialization step can be avoided by the user if the compiler has a startup library support.*

Note: `_SDA_BASE_` (`_SDA2_BASE_` follows the same strategy) *represents the start address of the sdata section (as defined into the linker file) + 0x8000: the start of the appropriate small data area section plus 32K.*

### 4.3.2 Core 0 Register initialization

Since in the SPC574K72/SPC57EM80 family the Core 0 is in Lock step, it needs its registers initialization before any use. In Lock Step mode (LSM), at power on, the two cores will contain different random data and if for example there is a store to the memory (e.g. stacked) it will cause a Lock Step error (see *Appendix C: Core0 registers initialization code*).

## 4.4 Enable Branch Target Buffer

The SPC574K72/SPC57EM80 Power Architecture core (e200zx) features a branch prediction optimization which can be enabled to improve the overall performance by storing the results of branches and using that to predict the direction of the future branches at the same location. To initialize it, the user needs to flash, invalidate the buffer and enable branch prediction.
This can be accomplished with a single write to the Branch Unit Control and Status Register (BUCSR) in the core.
Below is a scratch code to configure the BTB:

```
;#-----------------------------------------------------------------#
;# Flush and Enable BTB - Set BBFI and BPEN fields (BUCSR register) #
;#-----------------------------------------------------------------#
e_lis r3, 0x0
e_or2i r3, 0x0201
mtspr 1013, r3 ;#move content of register r3 into BUCSR register
```

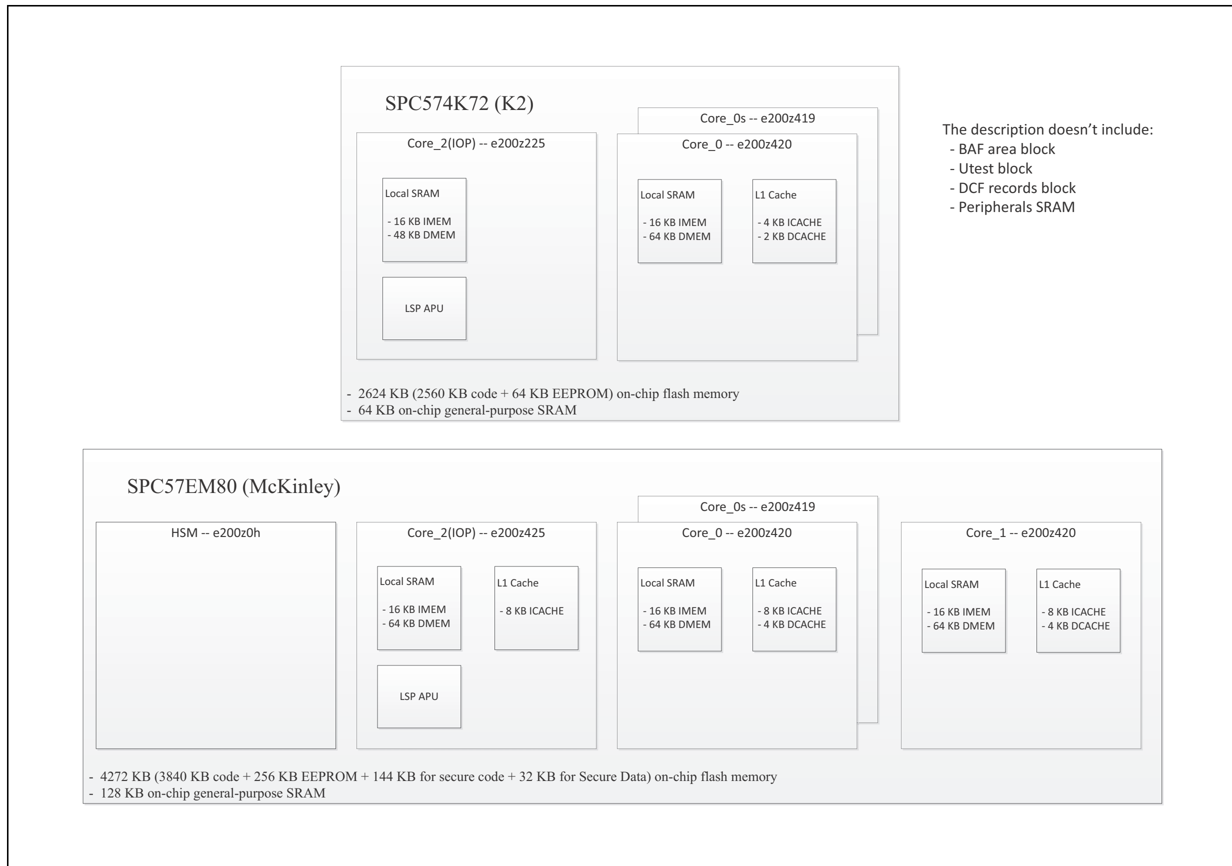
Note: *If the application modifies the instruction code in memory after this initialization procedure, the Branch Target Buffer may need to be flushed and re-initialized as it may contain a branch.*

Note: *It's possible to control the static prediction mechanism on a BTB miss through the BPRED field and the BTB allocation for a branch acceleration through BALLOC field.*

## 4.5 SRAM ECC Initialization

The on-chip general-purpose SRAM, as well as the local SRAM (Tightly-coupled memories internal to the cores) in the SPC574K72/SPC57EM80 family (as other SPC devices) has the ECC (Error Correction Code) protection.

**Figure 9. SPC574K72/SPC57EM80 memory and cores allocation**

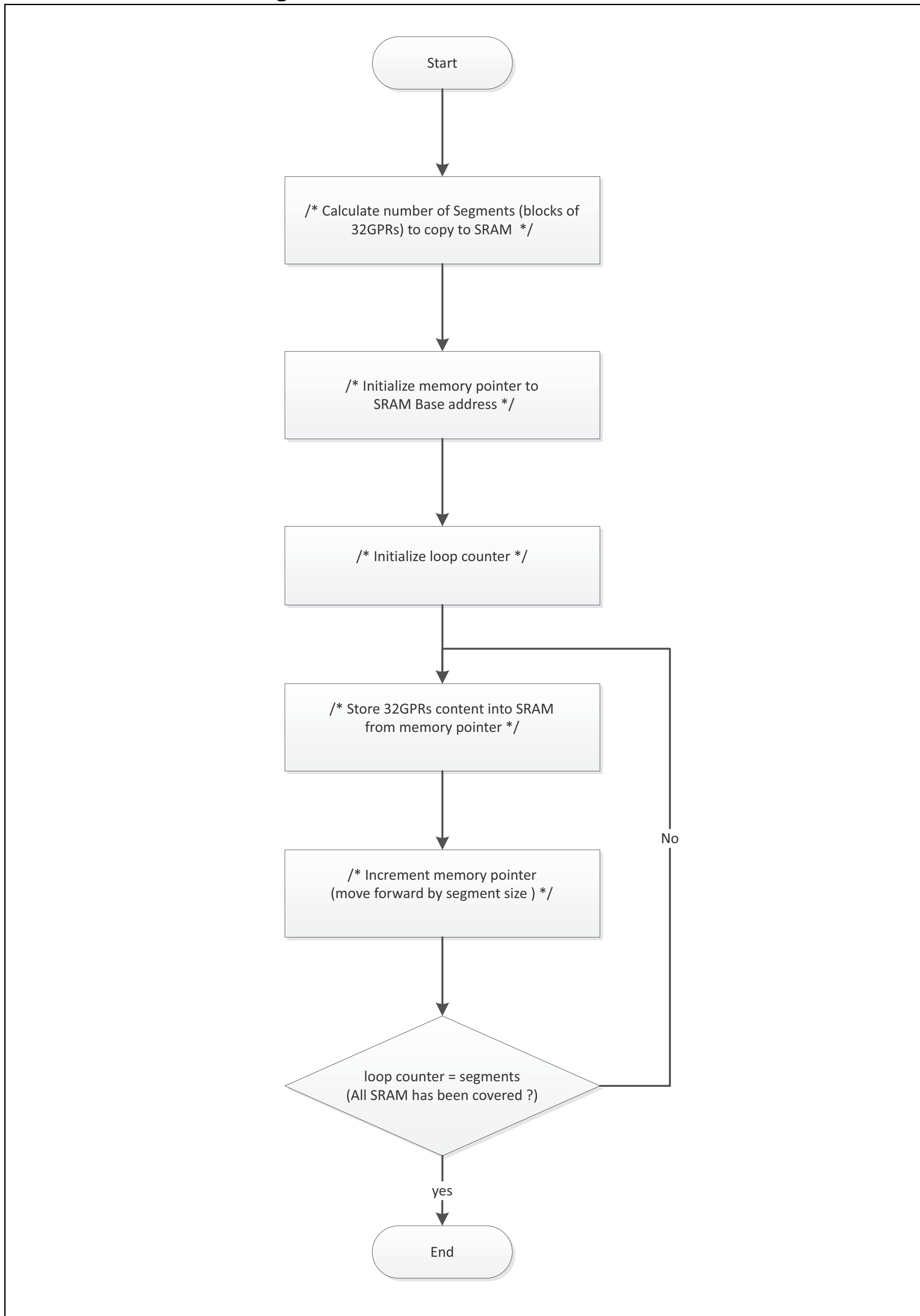



ECC checks are performed during the read portion of an SRAM ECC read/write (R/W) operation, and ECC calculations are performed during the write portion of a R/W operation.

Because the ECC bits can contain random data after the device is powered on, the SRAM must be initialized by executing 64-bit write operations prior to any read accesses[j] to avoid any ECC error and therefore an exception being raised (see *Figure 10: SRAM ECC Initialization flow*).

j. This is also true for implicit read accesses caused by any write accesses of less than 64 bits

**Figure 10.SRAM ECC Initialization flow**



Below the code to initialize the SRAM ECC:

```
;#/*=================================================================*/
;# macro to initialize sram_ecc
;# param size_value: ram size [bytes]
;# param addr_value: base address
.macro sram_ecc_init  size_value addr_value

;#********************** Initialise SRAM ECC ***************************/
;# Store number of 128Byte (32GPRs) segments in Counter
 e_lis r5, size_value@h;#/* Initialize r5 to size of SRAM (Bytes) */
 e_or2i r5, size_value@l;#/* */
 e_srwi r5, r5, 0x7;#/* Divide SRAM size by 128 */
 mtctr  r5;#/* Move to counter for use with "bdnz" */

;# Base Address of the Local SRAM
 e_lis r5, addr_value@h
 e_or2i r5, addr_value@l


;# Fill Local SRAM with writes of 32GPRs
 1:
    e_stmw r0,0(r5);#/* Write all 32 registers to SRAM */
    e_addi r5,r5,128;#/* Increment the RAM pointer to next 128bytes */
    e_bdnz 1b;#/* Loop for all of SRAM */

.endm
;#/*=================================================================*/
```

## 4.6 Environment Initializations

The Stack/Heap memory sections, the EABI registers (see *Section 4.3.1: EABI Register initialization*) must be initialized as well as constants and pre-initialized variables being copied from Flash to RAM (see *Appendix A: Copy Initialized Data*).
These initialization steps can be done by the user scratch or left to a pre-built compiler initialization script (see *Figure 11: Booting flow using GHS startup libraries*).
In any case these initialization steps are tightly coupled to the linker file and are compiler specific.

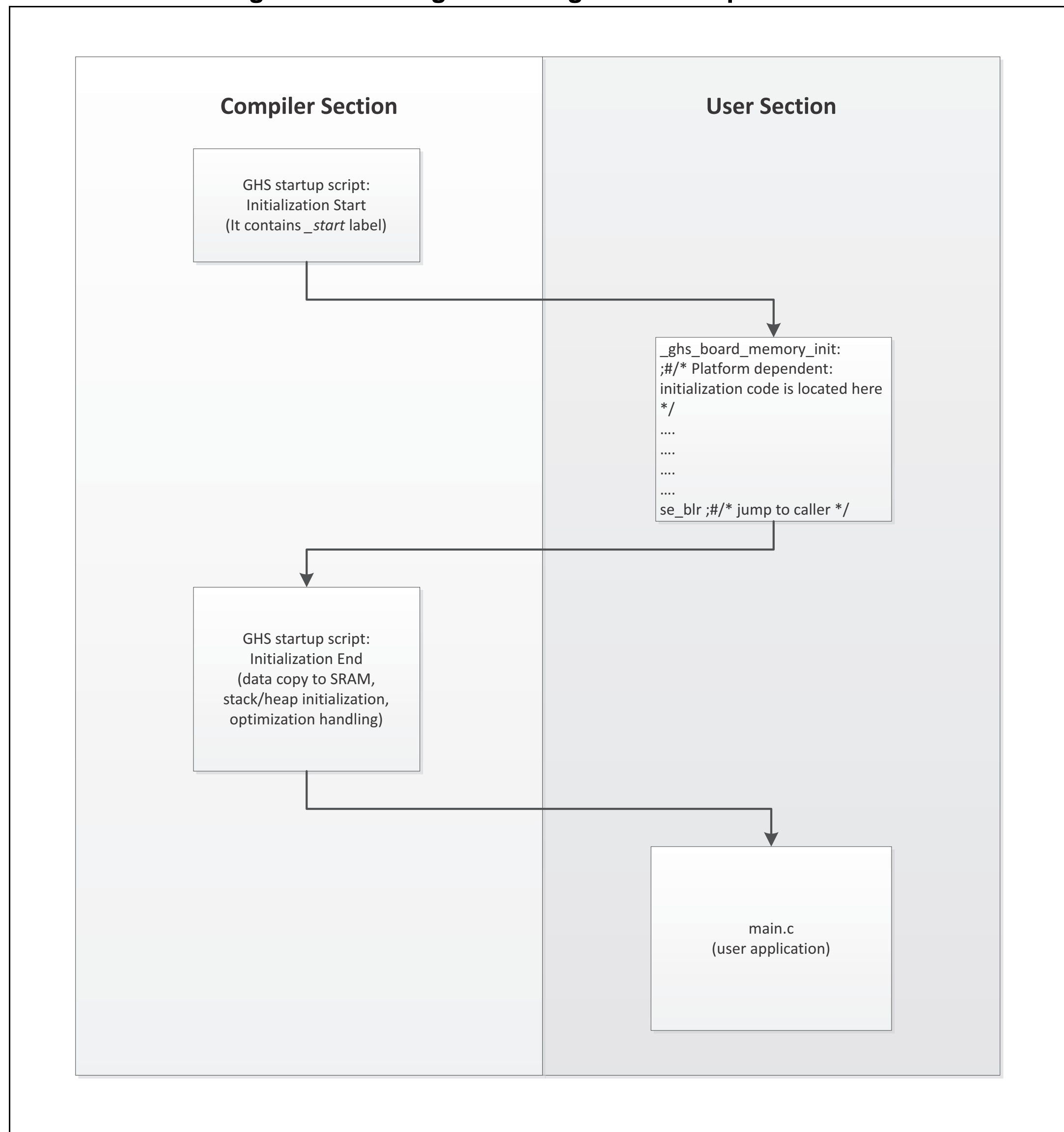
In this document the Green Hills compiler (GHS) support has been used in order to facilitate the code development and description. Since the boot-up is done from the IOP[k], the

k. The multi-core application described in this document consist on one binary images (unique environment for all the cores)

support is only valid for this core while the other cores have to do some minor initializations (see Section 4.3: Core registers initialization).

This means that the first instruction executed by the IOP (see Section 4.1: A valid Boot Header) is a GHS library instruction (see Figure 11: Booting flow using GHS startup libraries).

**Figure 11.Booting flow using GHS startup libraries**




## 4.7 XBAR Configuration

The crossbar switch connects bus masters and bus slaves using a hardware interconnect matrix. This structure allows all bus masters to access to the different bus slaves simultaneously with no interference while providing arbitration among the bus masters when they access to the same slave. A variety of bus arbitration methods and attributes may be

programmed on a slave-by-slave basis.
This devices family contains two crossbar switch modules:

- XBAR_0 - Computational Shell (Fast Crossbar)
- XBAR_1 - Peripheral Shell (Slow Crossbar)

*Figure 12* illustrates that they are connected via a Cross-lake, which is comprised of intelligent bridging bus gaskets.
These bus gaskets handle the traffic which crosses between the shells and do not require any interaction with the user. This dual-crossbar architecture allows all the masters to access to all slaves across the computational shell and the peripheral shell.

**Figure 12.Crossbar Switch diagram**

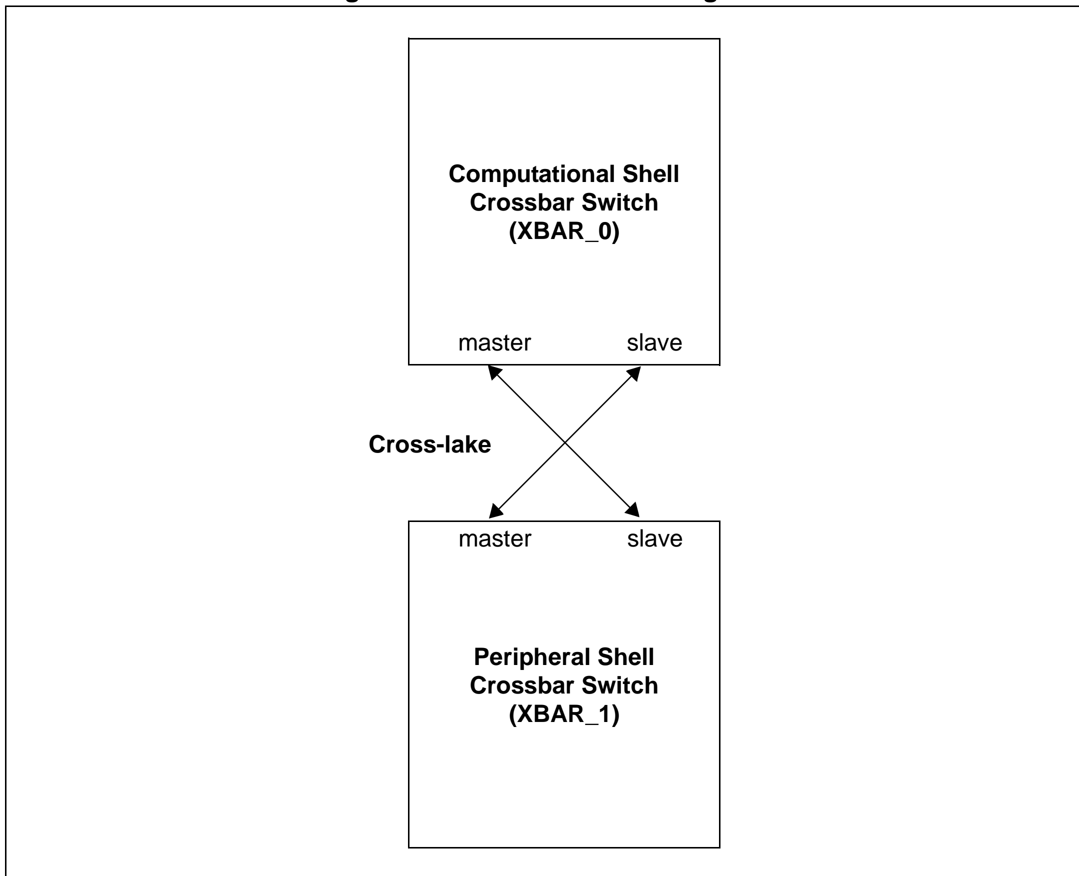
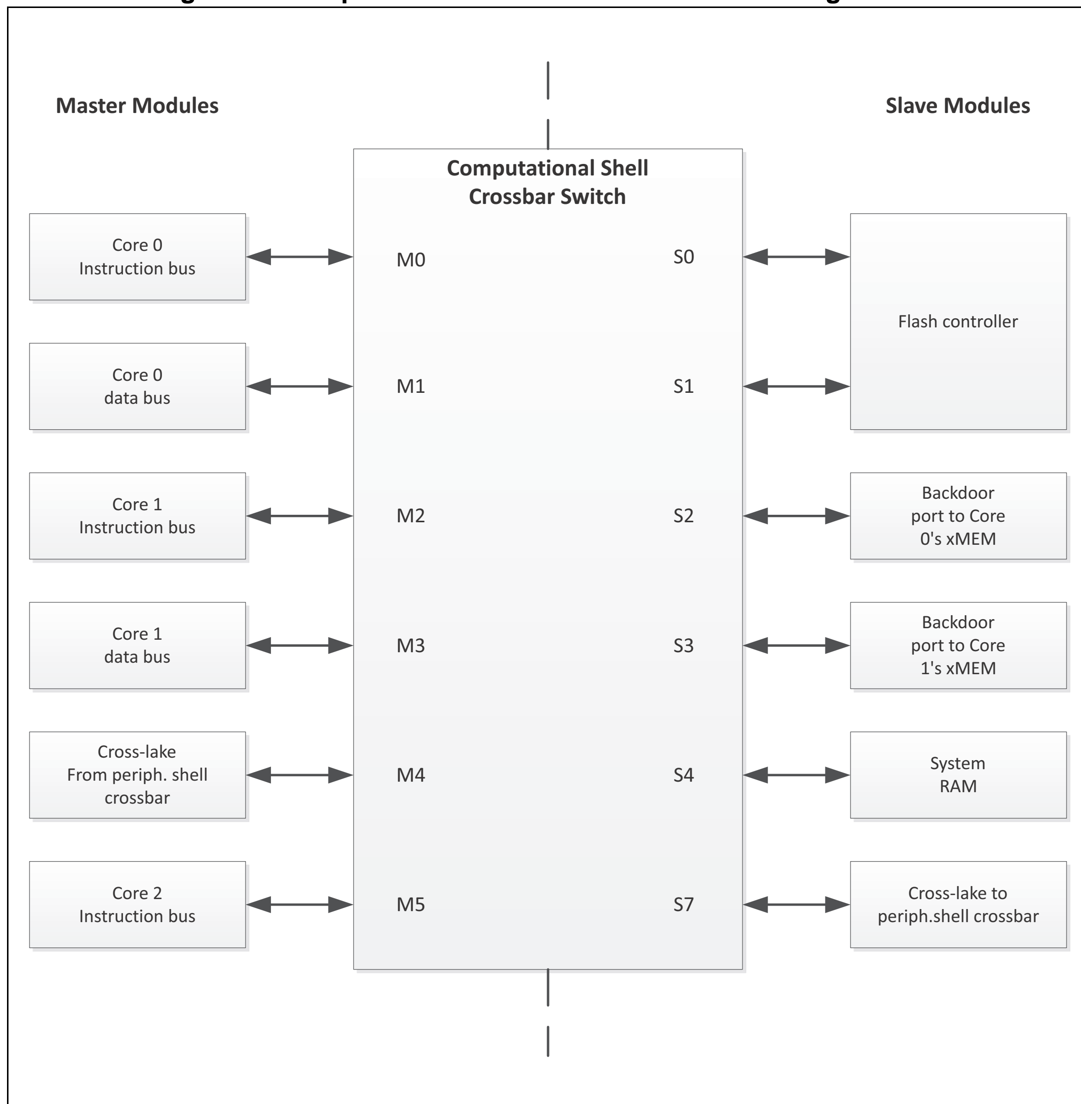


**Figure 13.Computational Shell's crossbar switch integration(l)**




*Note: Figure 13 shows the SPC57EM80 XBAR Computational shell (SPC574K72 doesn't have the Core 1, so that Slave 3 as well as Master 2 and Master 3 are not showed).*

### 4.7.1 Flash Memory Access

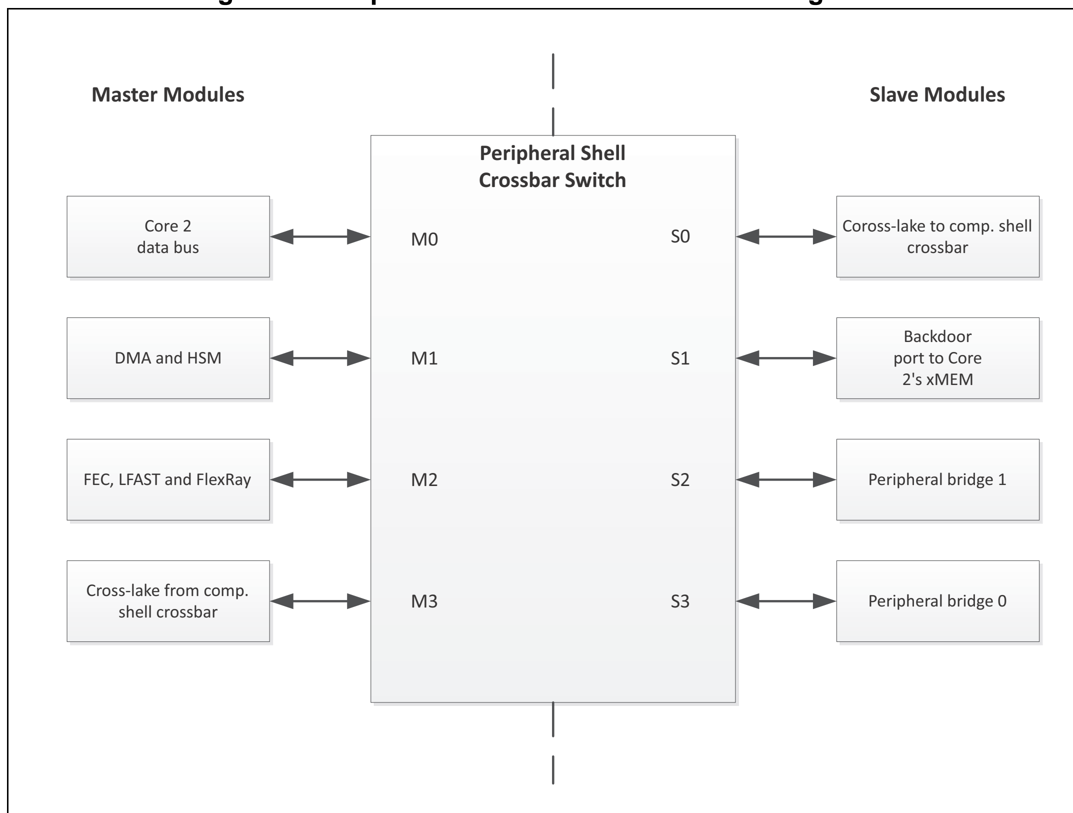
The access to the flash memory via slave ports 0 and 1 is routed based on the master so that the routing have been chosen to balance the data traffic in order to facilitate the core instruction fetches. Table 4 shows the flash memory port assignments.

l. Mx represent Master modules while Sx represent Slave modules

**Table 4. Flash memory port assignment**

| Flash memory slave port 0 | Flash memory slave port 1 |
|---|---|
| Core 0 Instruction bus (M0) | Core 0 data bus (M1) |
| Core 1 data bus (M3)<br>(only for the SPC57EM80) | Core 1 Instruction bus (M2)<br>(only for the SPC57EM80) |
| Peripheral shell crossbar switch (M4) | Core 2 Instruction bus (M5) |

**Figure 14.Peripheral Shell's crossbar switch integration**




### 4.7.2 XBAR Register configuration

The master priority assignments for the masters of both XBARs (readable through XBARn Priority Registers) are hardwired (0x0030_1425h for PRSn registers of XBAR_0 and 0000_3102h for the PRSn registers of XBAR_1).

The user can configure the slave behavior through the slave control registers (XBARx_CRSn):

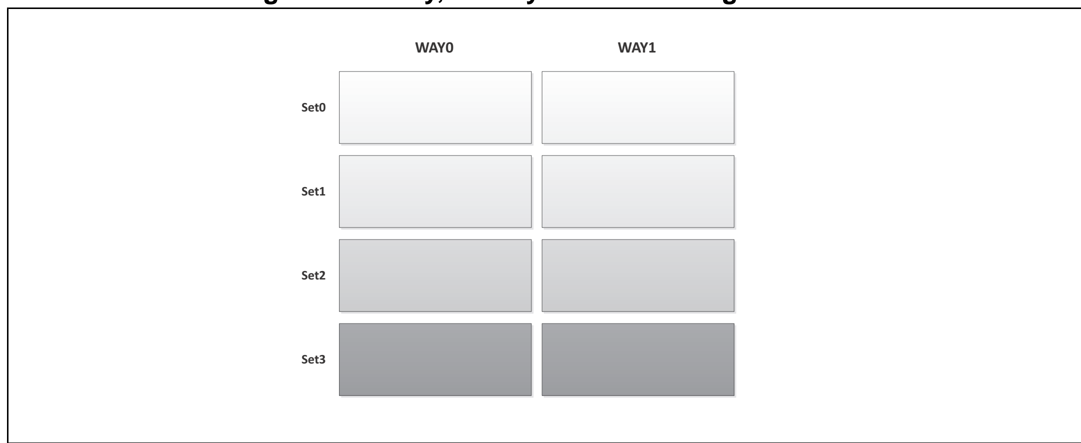
- Temporarily elevate some master requests to the slaves to high priority status
- Arbitration policy
- Parking control strategy

## 4.8 Memory Controllers configuration

### 4.8.1 Flash Controller configuration

The flash memory controller supports 2 64-bit AHB buses and a 256-bit read data interface to the flash memory array (see *E.1: Reference documents* for further details). Each AHB port contains a 4-entry, 2-way set-associative mini-cache (see *Figure 15: 2-way, 4-entry mini-cache organization*) as well as an associated controller that prefetches sequential lines of data from the flash arrays into the mini-cache (see *E.1: Reference documents*).
Each mini-cache entry is 256 bits in size, matching the code flash array page size and providing 256 bytes of high-speed local storage.

**Figure 15.2-way, 4-entry mini-cache organization**




Bus masters may be enabled or disabled from triggering prefetches, and triggering may be further restricted based on whether a read access is for instruction or data (Prefetched data is always loaded into the least-recently-used buffer). The user can choose several algorithms for a prefetch control which trade off the performance for power.
The prefetch strategy is configurable through the PFCR1 register (Platform Flash Configuration Register 1).
The arbitration of concurrent flash access requests from the two AHB ports of the flash memory controller and allocation[m] of the line read buffers are controlled via the PFCR3 control register (Platform Flash Configuration Register 3).

*Note: The user can controls executable access to the BAF (Boot Assist Flash) region of the flash through (BAF_DIS field of PFCR3). Once this field is set, attempted instruction accesses targeting the BAF region are aborted and terminated with a system bus error.*

Parameters affecting the transfers between the flash array and the cache, such as read wait states, are not optimal out of reset and for the best performance should be configured for desired target system clock frequency.

m. The buffers can be organized as a “pool” of available resources (with both ways within a given set) or with a fixed partition between ways allocated to instruction or data accesses. For the fixed partitions, ways 0 is allocated for instruction fetches and way1 for data accesses.

When modifying characteristics for a memory, such as in this example, it is good practice not to execute code in the same memory that is having its characteristics modified. Hence the code to modify the flash performance parameters of the module's configuration register will be executed from the SRAM.

### 4.8.2 SRAM wait State

The user, through the configuration of PFCR1 (Platform RAM configuration Register 1), can specify operation of the RAM controller[n] (see *Section E.1: Reference documents*).

The configuration of this register allows the user to control the AHB arbitration access as well as the response time (number of cycles taken for a RAM access) for a 64-bit read burst for the both ports. It is also possible to configure an additional cycle latency on a AHB response of the RAM controller on reads.

*Note: The PFCR1 register is accessible only in supervisor mode so that any access in user mode returns a transfer error.*

## 4.9 Mode Entry Module: Configuration

The SPC574K72/SPC57EM80 device family (as between other SPC56xx devices) has several operating modes (see *E.1: Reference documents*).

The MC_ME controls the chip mode and mode transition sequences in all functional states (see *Figure 16: Mode Entry Diagram*). It also contains the configuration, control and status registers accessible for the application.

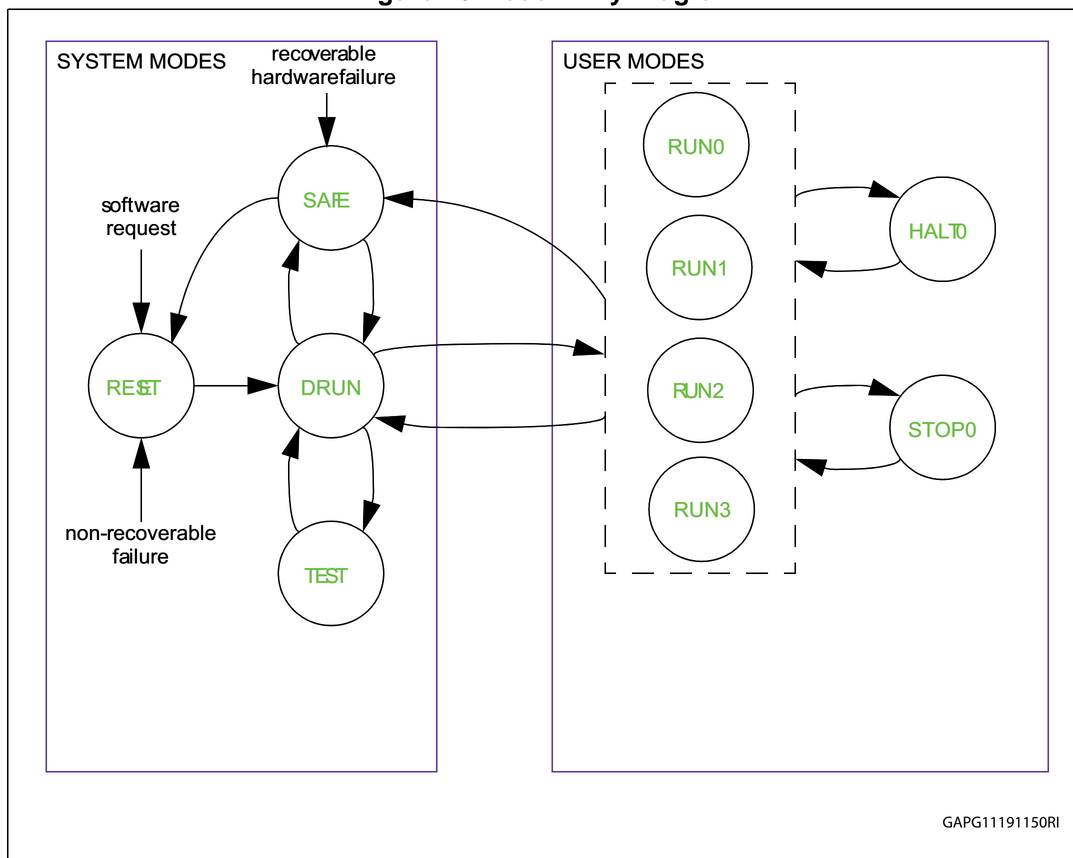
Each mode is configurable and can define a policy for energy and processing power management to fit particular system requirements. An application can easily switch from one mode to another depending on the current needs of the system.

It is important to highlight that this module has an additional core clock gating and boot address control based on the ME_CCTLn and ME_CADDRn registers.

*Note: Out of reset the device leaves the reset mode and enters DRUN mode.*

n. The RAM controller supports two 64-bit AHB interfaces and a 64-bit RAM array interface. The primary AHB port provides a connection to the platform crossbar for direct RAM access from the various crossbar masters.The backdoor AHB port provides a connection from the flash controller to facilitate the calibration overlay access.

Figure 16.Mode Entry Diagram




In order to use all of the available modes, they must be enabled in the Mode Enable register Below a scratch code that shows how to configure it:

```
#define DRUN_MODE 0x3
...
MC_ME.ME.R = 0x000005E2;          /* Enable all modes */

/* MC_ME.DRUN_MC.R not yet configured...IRC Osc by default */

/* Setting RUN Configuration Register ME_RUN_PC[0] */
MC_ME.RUN_PC[0].R = 0x000000FE;/* Peripheral ON in every mode */
MC_ME.RUN_PC[1].R = 0x000000FE;/* Peripheral ON in every mode */
MC_ME.RUN_PC[2].R = 0x000000FE;/* Peripheral ON in every mode */
MC_ME.RUN_PC[3].R = 0x000000FE;/* Peripheral ON in every mode */
MC_ME.RUN_PC[4].R = 0x000000FE;/* Peripheral ON in every mode */
MC_ME.RUN_PC[5].R = 0x000000FE;/* Peripheral ON in every mode */
MC_ME.RUN_PC[6].R = 0x000000FE;/* Peripheral ON in every mode */
MC_ME.RUN_PC[7].R = 0x000000FE;/* Peripheral ON in every mode */

/* Turn On XOSC - PLL's remain off */
MC_ME.DRUN_MC.R = 0x00130020;    /* Enable the XOSC */

/* Trigger DRUN mode Transision */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x00005AF0);/* Mode & Key */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x0000A50F); /* Mode & Key inverted */
```

```
/* Wait for mode entry to complete */
while(MC_ME.GS.B.S_MTRANS == 1);
/* Check DRUN mode has been entered */
while(MC_ME.GS.B.S_CURRENT_MODE != # DEFINE DRUN_MODE 0x3); /
```

Note: *In the device configuration flow, in order to take effect all changes in all clock and mode configuration, a mode transition must be done.*
*For further information look at Mode Entry Module chapter in the device Reference Manual (see E.1: Reference documents).*

### 4.9.1 Core Control Register Configuration

The user alternatively to the boot header can control the core behavior through the mode entry module and in particular with the registers ME_CCTLx and ME_CADDRx.
ME_CCTLx register controls whether core_x is disabled or running during run modes.

**Table 5. Core Control Register x (ME_CCTLx)**

Access: User read, Supervisor read/write, Test read/write

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | 0 | 0 | 0 | 0 | 0 | STOP0 | 0 | HALT0 | RUN3 | RUN2 | RUN1 | RUN0 | DRUN | SAFE | TEST | RESET |
| W | | | | | | | | | | | | | | | | |

ME_CADDRx has the core_x boot address and sets (through RMC field) whether this core resets when a mode change configures it to run in the new mode.

Note: *The chip configuration and boot mode determine the reset value of ME_CADDRx[ADDR].*

#### Core Control Registers (ME_CADDR0)

**Table 6. Core Address Register x (ME_CADDRx)**

Access: User read, Supervisor read/write, Test read/write

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R/W | ADDR[31:16] | | | | | | | | | | | | | | | |
| Reset | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |

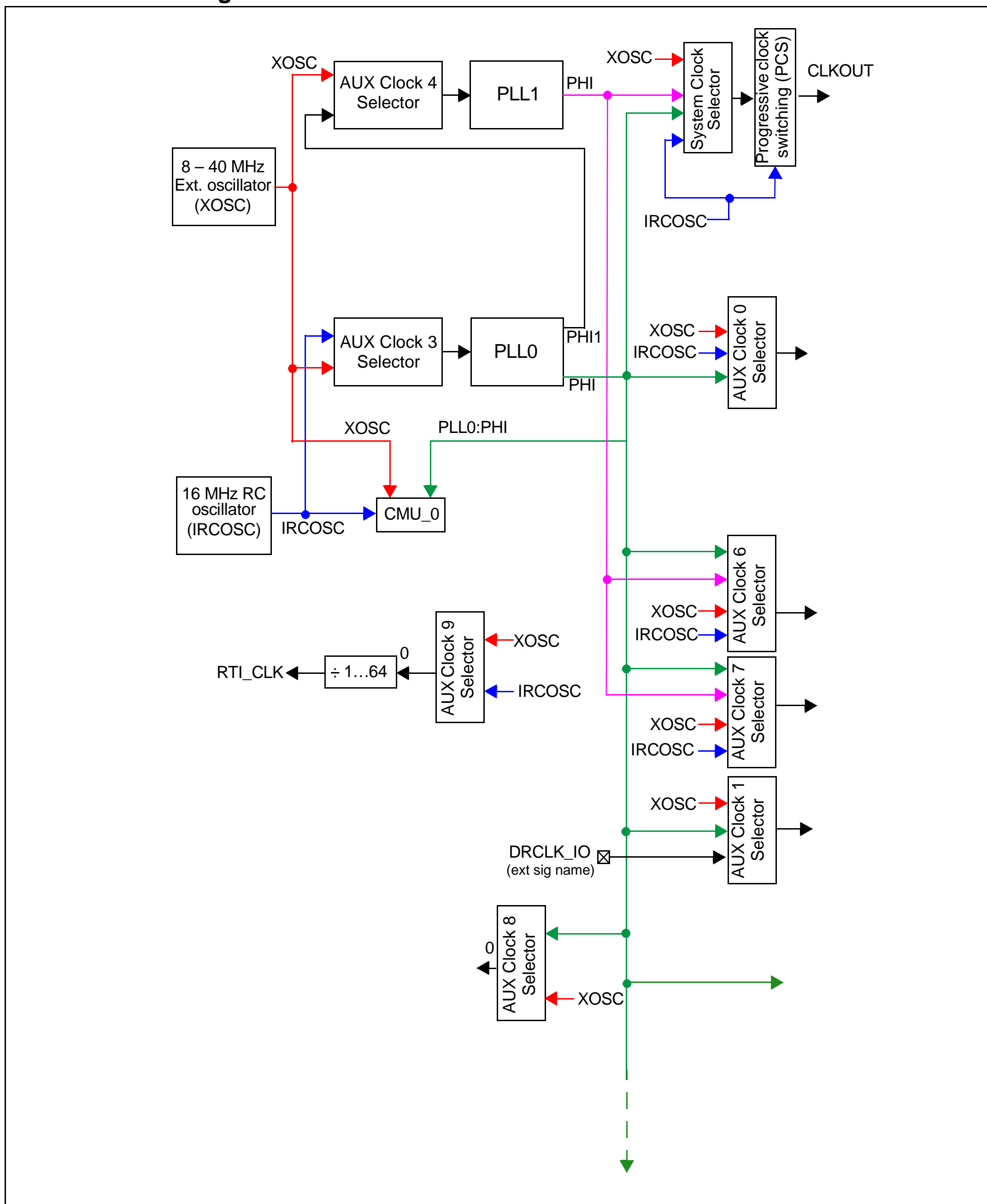
| | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | ADDR[15:2] | | | | | | | | | | | | | | 0 | RMC |
| W | | | | | | | | | | | | | | | | |
| Reset | X | X | X | X | X | X | X | X | X | X | X | X | X | X | 0 | 0 |

Note: *These registers cannot be written after a mode change request until the mode transition completes (i.e., while ME_GS[S_MTRANS] = ‘1’). Write access to this register during this time results in assertion of the ICONF_CC flag in the ME_IS register.*

## 4.10 Clock and PLL configuration

The devices have several functional blocks that run at different frequency and for this reason before the clocks are changed the user has to take some constraints into account (the individual clock dividers must be set accordingly) to allow the cores and each IP inside the devices to run up to the desired frequency (see also System clock frequency limitations chapter listed into *E.1: Reference documents*).
The system clock (CLKOUT) for the computational shell (see *Figure 3: SPC574K72 Block diagram* and *Figure 4: SPC57EM80 Block diagram*): which includes the main CPUs as well as all modules that run in the high speed domain), memories, and debug logic is independent from the peripheral clocks.

**Figure 17.SPC574K72/SPC57EM80 Clock Generation**




This family devices boot from the internal 16 MHz RC oscillator (IRCOSC), and use this as a backup clock in the event of a PLL or oscillator failure (if the backup clock is enabled). See the “Mode Entry Module (MC_ME)” chapter for details.

There are three possible ways to provide the source clock:

- External oscillator
- External crystal
- Internal 16 MHz RC oscillator

From one of these input sources, the internal clocks are generated from one of the two

PLLs, PLL0 and PLL1, using the PLL0_PHI and PLL1_PHI outputs, respectively.
These two clocks, along with the XOSC and IRCOSC, can be selected to drive the system peripherals depending on the configuration of the Auxiliary Clock Selectors in the Clock Generation Module (MC_CGM).
The PHI1 output of PLL0 can also be used as the clock source for PLL1.
Each of the outputs of the module clock selectors have individual dividers that allow for more clock frequency granularity for a given peripheral (see *Appendix D: Clocks and PLLs Initialization example* for reference).

## 4.11 Cache configuration

The e200zx core processors used in this product family have caches with the following characteristics:

- 2 way set-associative
- 32-byte line size
- Writethrough Support
- 8-entry Store Buffer
- 64-bit data
- 32-bit address

The caches, as it is easy to understand, speed up the device performances because they decouple processor performance from system memory performance (see *E.1: Reference documents*).
Enabling the instruction cache will reduce execution time for the remainder of the initialization procedure and, of course, improve application execution speed later on.
It's up to the user to choose the appropriate time to enable it based on his own requirements taking into account for example the consumption picture versus the increased performances.

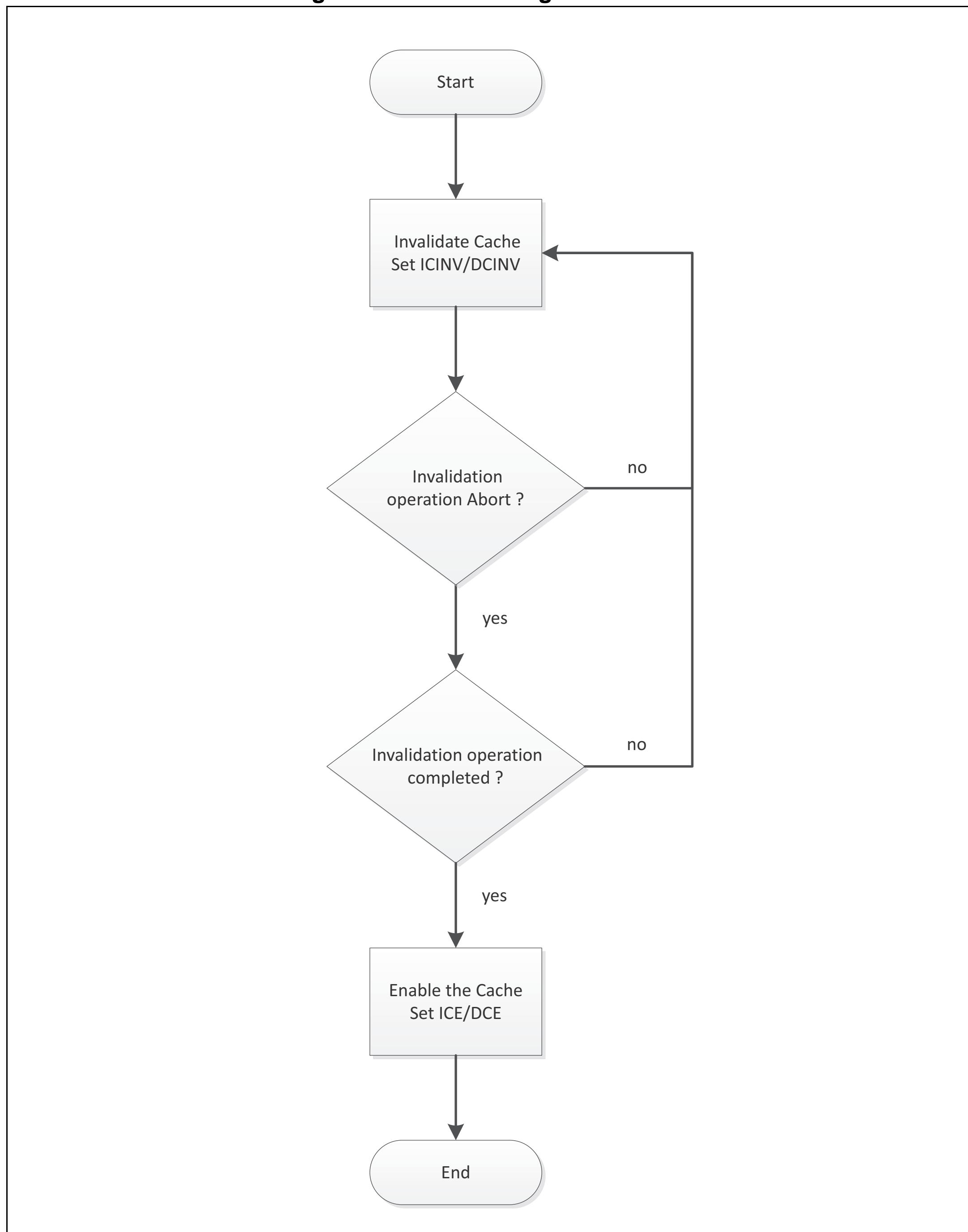
*Note: Enabling the instruction cache after the PLL is programmed, can increase peak current draw during the setup routine, so it is recommended that the cache be enabled first. If the user chooses this strategy it is suggested, if the code is executed by the internal flash, to inhibit the flash space, initially, after the PLL programming. The reason for this is the polling loop that tests PLL lock. With the cache enabled this loop can execute very quickly and draw more current. There is not a real need for the polling loop to execute that quickly at this point, so the user should inhibit the flash region from caching and then enable the cache.*

The core instruction and data caches are enabled through the L1 Cache Control and Status Registers (L1CSR0 and L1CSR1). The instruction cache is invalidated and enabled by setting the ICINV and ICE bits in the L1CSR1 register (in similar way, the invalidation and the enabling of the data cache is done through the DCINV and DCE bits in the L1CSR0 register).

*Note: The cache invalidates the operation that takes some time and can be interrupted or aborted (see Appendix B: Cache Configuration source code).*

A robust cache enables the configuration flow (see *Figure 18.: Cache configuration flow*) checks to ensure the invalidation has successfully been completed and if not, retries the operation before enabling the cache.

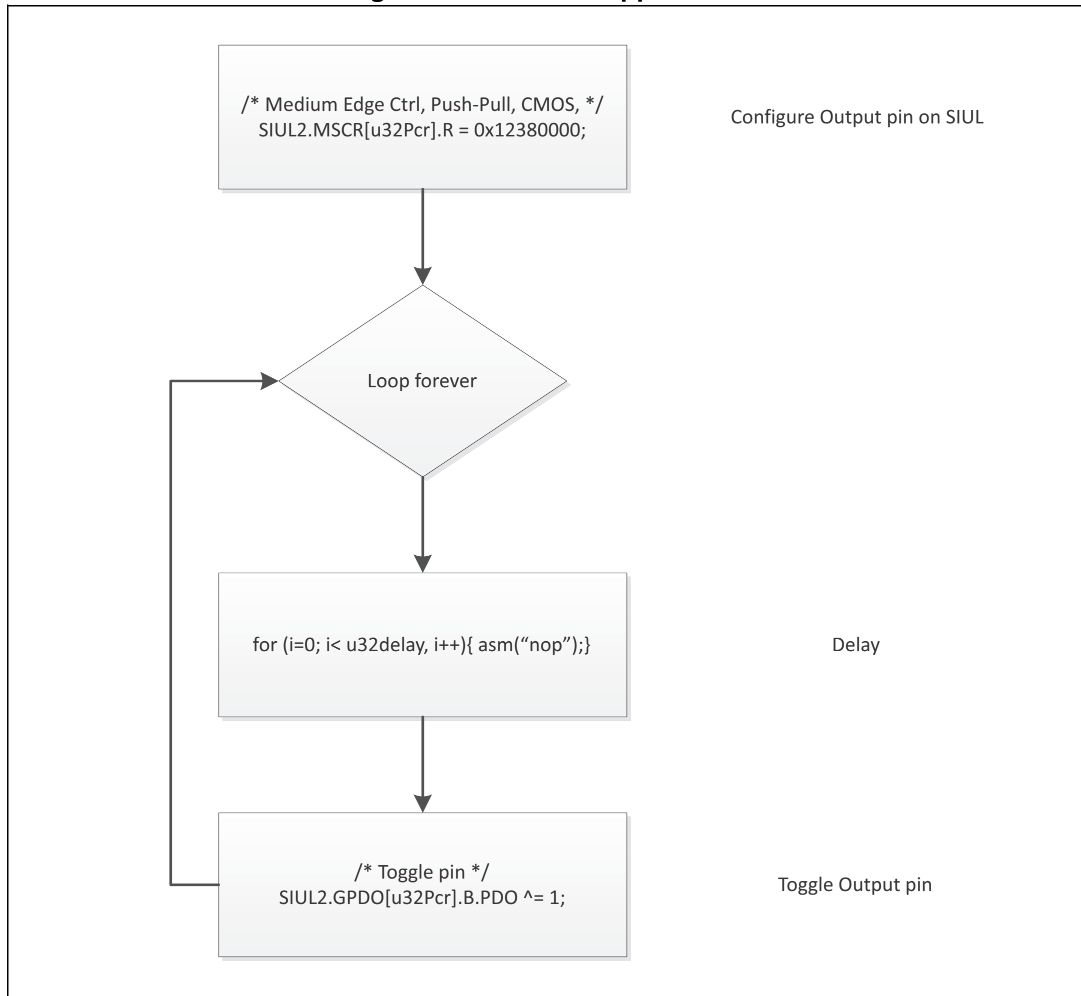
**Figure 18. Cache configuration flow**



# 5 Blink LED application

As soon each core has ended its own initialization flow the user can finally execute his application (see *Figure 19: Blink LED application*).

**Figure 19.Blink LED application**



# Appendix A Copy Initialized Data

When the applications boot from flash, the program image stored in flash will contain the various data segments created by the C compiler and linker. Initialized read-write data must be copied by the user from read-only flash to read-writable SRAM before execution flow reaches the application. Many compilers allow the user to use its own startup libraries to do this copy.

The below code shows the steps the user has to do if he doesn't want to use any compiler support.

```
;#================================================
;# Initialized Data - ".data"
;#================================================
DATACOPY:
e_lis r9, __DATA_SIZE@ha ;# Load upper SRAM load size
e_or2i r9, __DATA_SIZE@l ;# Load lower SRAM load size into R9
e_cmp16i r9,0 ;# Compare to see if equal to 0
e_beq SDATACOPY ;# Exit cfg_ROMCPY if size is zero
mtctr r9 ;# Store no. of bytes to be moved in counter

e_lis r10, __DATA_ROM_ADDR@h ;# Load address of first SRAM load into R10
e_or2i r10, __DATA_ROM_ADDR@l ;# Load lower address of SRAM load into R10
e_subi r10,r10, 1 ;# Decrement address
e_lis r5, __DATA_SRAM_ADDR@h ;# Load upper SRAM address into R5
e_or2i r5, __DATA_SRAM_ADDR@l ;# Load lower SRAM address into R5
e_subi r5, r5, 1 ;# Decrement address

DATACPYLOOP:
e_lbzu r4, 1(r10) ;# Load data byte at R10 into R4
e_stbu r4, 1(r5) ;# Store R4 data byte into SRAM at R5
e_bdnz DATACPYLOOP ;# Branch if more bytes to load from ROM

;#================================================
;# Small Initialised Data - ".sdata"
;#================================================
SDATACOPY:
e_lis r9, __SDATA_SIZE@ha ;# Load upper SRAM load size
e_or2i r9, __SDATA_SIZE@l ;# Load lower SRAM load size into R9
e_cmp16i r9,0 ;# Compare to see if equal to 0
e_beq ROMCPYEND ;# Exit cfg_ROMCPY if size is zero
mtctr r9 ;# Store no. of bytes to be moved in counter

e_lis r10, __SDATA_ROM_ADDR@h ;# Load address of first SRAM load into R10
e_or2i r10, __SDATA_ROM_ADDR@l ;# Load lower address of SRAM load into R10
e_subi r10,r10, 1 ;;# Decrement address

e_lis r5, __SDATA_SRAM_ADDR@h ;# Load upper SRAM address into R5
e_or2i r5, __SDATA_SRAM_ADDR@l ;# Load lower SRAM address into R5
e_subi r5, r5, 1 # Decrement address

SDATACPYLOOP:
e_lbzu r4, 1(r10) ;# Load data byte at R10 into R4
e_stbu r4, 1(r5) ;# Store R4 data byte into SRAM at R5
```

```
e_bdnz SDATACPYLOOP ;# Branch if more bytes to load from ROM

ROMCPYEND:
```

__DATA_SIZE, __DATA_ROM_ADDR, __DATA_SRAM_ADDR, __SDATA_SIZE, __SDATA_ROM_ADDR, __SDATA_SRAM_ADDR have to be defined in the linker file

# Appendix B Cache Configuration source code

```
;# macro to allow immediate register load to be done more easily
.macro e_lwi  register value
 e_lis register, value@h
 e_or2i register, value@l
.endm

.globl      ICACHE_Enable
.globl      DCACHE_Enable

.section    .vletext, axv
.vle
.align 4

;#.fsize 0
.type ICACHE_Enable,@function
;#==================================================
;#  Function to enable the instruction cache
;#==================================================
ICACHE_Enable:
;#
;#-----------------------------------------#
;# Invalidate Instruction Cache - Set ICINV #
;# bit in L1CSR1 Register #
;#-----------------------------------------#
icache_cfg:
e_li r5, 0x2

msync
se_isync
mtspr 1011,r5 ;#mtl1csr1
se_isync
;#-----------------------------------------#
;# Mask out ICINV and ICABT to see if #
;# invalidation is complete (i.e. ICINV=0, #
;# ICABT=0) #
;#-----------------------------------------#
e_li r7, 0x4;#Load ICABT mask into R8
```

```
e_li r8, 0x2;#Load ICINV mask into R8
e_lwi r11, 0xFFFFFFFB;#Load ICABT clear mask into r11

icache_inv:
msync
mfspr r9, 1011;#mfl1csr1: Read L1CSR1 register, store into r9

and.  r10, r7, r9;#check for an ABORT of the cache invalidate operation
e_beq icache_no_abort

and.  r10, r11, r9;#If abort detected, clear ICABT bit and re-run
invalidation

msync
se_isync
mtspr 1011, r10;#mtl1csr1
se_isync

e_b icache_cfg

icache_no_abort:
;#-----------------------------------------#
;# Check that invalidation has completed -  #
;# (ICINV=0). Branch if invalidation not#
;# complete. #
;#-----------------------------------------#
and.  r10, r8, r9
e_bne icache_inv

;#-----------------------------------------#
;# Enable cache the ICache by performing a #
;# read/modify/write of the ICE bit in the #
;# L1CSR1 register #
;#-----------------------------------------#
msync
mfspr r5, 1011;#mfl1csr1

e_ori   r5, r5, 0x0001;#enable ICache (ICE=1)

msync
```

```
se_isync
mtspr 1011, r5;#(mtl1csr1) Write R5 to L1CSR1 register
se_isync

se_blr

.scall ICACHE_Enable, __leaf__
.size ICACHE_Enable,$-ICACHE_Enable
```

# Appendix C Core0 registers initialization code

```
;#/*===================================================================*/
;# macro to init registers to a know value

.macro REG_init
;# GPR's 0-31
 e_li   r0, 0
 e_li   r1, 0
 e_li   r2, 0
 e_li   r3, 0
 e_li   r4, 0
 e_li   r5, 0
 e_li   r6, 0
 e_li   r7, 0
 e_li   r8, 0
 e_li   r9, 0
 e_li   r10, 0
 e_li   r11, 0
 e_li   r12, 0
 e_li   r13, 0
 e_li   r14, 0
 e_li   r15, 0
 e_li   r16, 0
 e_li   r17, 0
 e_li   r18, 0
 e_li   r19, 0
 e_li   r20, 0
 e_li   r21, 0
 e_li   r22, 0
 e_li   r23, 0
 e_li   r24, 0
 e_li   r25, 0
 e_li   r26, 0
 e_li   r27, 0
 e_li   r28, 0
 e_li   r29, 0
 e_li   r30, 0
 e_li   r31, 0
```

```
;# Init any other CPU register which might be stacked (before being used).
 mtspr 1,r1;#XER
 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 IVPR, r1
 mtspr USPRG0, r1
 mtspr 62, r1       ;#ESR
 mtspr 8,r31 ;#LR
.endm
```

# Appendix D Clocks and PLLs Initialization example

```
#define DRUN_MODE 0x3
#define DIVIDEBY1 0x0
#define DIVIDEBY2 0x1
#define DIVIDEBY3 0x2
#define SELCTL_16MHz_IRC 0x0/* Internal RC Osc */
#define SELCTL_CRYSTAL_OSC 0x1/* External Osc */
#define SELCTL_PLL0  0x2 /* PLL0 PHI */
#define SELCTL_PLL0_PH1 0x3 /* PLL0 PHI1 */
#define SELCTL_PLL1 0x4     /* PLL1 PHI*/
....
OUTCLK_Init();/* Configure Output Clocks */
/* Route XOSC to the PLL's - IRC is default */
MC_CGM.AC3_SC.B.SELCTL = SELCTL_CRYSTAL_OSC;/* Connect XOSC to PLL0 */
/* Trigger DRUN mode Transision */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x00005AF0);/* Mode & Key */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x0000A50F); /* Mode & Key inverted */
/* Wait for mode entry to complete */
while(MC_ME.GS.B.S_MTRANS == 1);
/* Check DRUN mode has been entered */
while(MC_ME.GS.B.S_CURRENT_MODE != DRUN_MODE);

PLLDIG.PLL0CR.B.LOLRE = 0; /*Ignore loss of lock. Reset not asserted.*/
PLL0_Init(pll0_clk);/* Configure PLL0(Init of FMPLL configuration regs) */

PLLDIG.PLL1CR.B.LOLRE = 0; /*Ignore loss of lock. Reset not asserted.*/
PLL1_Init(_SYSCLK_);/* Configure PLL1(Init of FMPLL configuration regs)*/

/* Set the System Clock */
/* ME_DRUN_MC - enable XOSC and PLLs - PLL1 is sysclk */
MC_ME.DRUN_MC.R = 0x001300F4;

/* Trigger DRUN mode Transision */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x00005AF0);/* Mode & Key */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x0000A50F);/* Mode & Key inverted */

/* Wait for mode entry to complete */
while(MC_ME.GS.B.S_MTRANS == 1);
```

```
/* Check DRUN mode has been entered */
while(MC_ME.GS.B.S_CURRENT_MODE != DRUN_MODE);

/* wait for PLL to lock - will not lock before DRUN re-entry */
while(PLLDIG.PLL0SR.B.LOCK == 0) { asm("nop");  };
while(PLLDIG.PLL1SR.B.LOCK == 0) { asm("nop");  };

MC_CGM.SC_DC[2].B.DIV = bridge_divider; /* PBRIDGE Clock Divider */
MC_CGM.SC_DC[1].B.DIV = DIVIDEBY2;/* Slow crossbar Clock Divide by 2 */
MC_CGM.SC_DC[0].B.DIV = DIVIDEBY1;/* Fast crossbar Clock Divide by 1 */

/* Enable and configure Aux clocks */

/* AUX Clock Selector 0 */
MC_CGM.AC0_SC.B.SELCTL = SELCTL_PLL0;/* PLL0 PHI */
/* Enable Auxilliary Clock 0 divider 0 (general peripheral clock) */

MC_CGM.AC0_DC[0].B.DE = 1; /* Enabled */
MC_CGM.AC0_DC[0].B.DIV = DIVIDEBY2;/* Divide by 2 */
/*....
....Configure all dividers
*/
/* Trigger DRUN mode Transision */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x00005AF0);/* Mode & Key */
MC_ME.MCTL.R = (DRUN_MODE << 28 | 0x0000A50F); /* Mode & Key inverted */
/* Wait for mode entry to complete */
while(MC_ME.GS.B.S_MTRANS == 1);
/* Check DRUN mode has been entered */
while(MC_ME.GS.B.S_CURRENT_MODE != DRUN_MODE);
```

# Appendix E Further Information

## E.1 Reference documents

1. *SPC574Kxx - 32-bit Power Architecture® based MCU for automotive applications* (RM0334, DocID023671)
2. *SPC57EM80xx - 32-bit Power Architecture® based MCU with up to 4 Mbyte Flash and 304 Kbyte RAM memories* (RM0314, DocID022530)
3. e200z4 Power Architecture™ Core Reference Manuals

## E.2 Acronyms and abbreviations

**Table 7. Acronyms and abbreviations**

| Terms | Meaning |
|---|---|
| BAM | Boot Assist Module |
| BAF | Boot Assist Flash |
| BTB | Branch Target Buffer |
| CMPU | Core Memory Protection Unit |
| CPU | Central Processing Unit |
| DCache | Data Cache |
| DCF | Device Configuration Format |
| DMA | Direct Memory Access |
| DMEM | Data Memory (internal to the core) |
| HSM | Hardware Security Module |
| ECC | Error Correcting Code |
| FCCU | Fault Collection and Control Unit |
| GPIO | General purpose input/output |
| GHS | Green Hills (Compiler) |
| ICache | Instruction Cache |
| IMEM | Instruction Memory (internal to the core) |
| IOP | I/O Processor |
| 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 |

**Table 7. Acronyms and abbreviations**

| Terms | Meaning |
|---|---|
| SIPI | Serial Interprocessor Interface |
| SoC | System on Chip |
| SoR | Sphere of Replication |
| SSCM | System status and configuration module |
| ST | STMicroelectronics |
| STCU2 | Self-Test Control Unit |
| SWT | Software Watchdog Timer |
| TCM | Tightly-Coupled Memory |
| XBAR | Crossbar Switch |

# Revision history

**Table 8. Revision history**

| Date | Revision | Changes |
|---|---|---|
| 19-Nov-2013 | 1 | Initial release. |
| 27-Mar-2014 | 2 | Modified *Table 1* and *Table 2*. |

**Please Read Carefully:**

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

**UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.**

**ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.**

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2014 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

**www.st.com**