AN5010
Application note

SPC570Sx Getting Started

Introduction

The SPC570Sx devices family is built on Power Architecture® technology and is targeted for ABS and airbag applications that require a high safety integrity level (ISO 26262 for ASIL-D safety integrity).
In order to minimize additional software and module level features to reach this target, an on-chip redundancy is offered for the critical components (see Functional Safety chapter of RM) of the microcontroller by an e200z0Hn2p core, a DMA controller and an interrupt controller in a delayed lockstep configuration, a crossbar bus system, a memory protection unit, a fault collection and control unit (FCCU), Flash and SRAM with end-to-end error correction coding (ECC) and End to end (E2E) protection on the data path and memory ECC.
This family operates up to 80MHz and offers a high performance processing power specially 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 SPC570Sx devices family from power-up to the code execution.
A development flow is described throughout the application note to explain the steps.

It is intended that this application note is read along with the SPC570Sx reference manual, that can be obtained from the STMicroelectronics website at http://www.st.com (see *Section D.1: Reference documents*).

Contents

List of tables

List of figures

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 out of internal Flash.
This family (SPC57x) superseded the previous one (SPC56x): It is built on 55nm technology and indeed apart from the technological point of view (55nm versus the older 90nm) it has been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 (see *Section D.1: Reference documents*).
SPC570Sx by default is configured so to replicate its safety-relevant processing elements (Core, DMA/DMAMUX, Interrupt Controller) and compares its operation in lockstep (see *Section D.1: Reference documents*).
Even if device works in lockstep (LockStep Mode or LSM), from an application perspective the device behaves as a single core but the user has to take care of its initialization (core registers, SRAM) to take into account safety architecture peculiarities.
The user can do these initialization by scratch or by using compilers support (through the compiler property libraries) (see *Figure 8: Booting flow using GHS startup libraries*) that allows the user to avoid some core register initializations (see also *Section 4.4: Core registers initialization*) as well as the data copy of initialized data from the Flash to SRAM (see *Section 4.7: Environment Initializations*) before the user code execution starts.

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 an 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 it doesn't include any special memory handling like memory protection configurations (see *Section D.1: Reference documents*).

2 Family architecture

Two identical e200z0Hn2p processors (see *Figure 1: SPC570Sx Block diagram*) are used: the main core, which executes the software application, and the safety core, linked to the main core in delayed lock-step mode for ASIL-D safety requirements.
The e200z0h processor family is a CPU core that it implement cost-efficient versions of the Power Architecture®.
These processors are designed for deeply embedded control applications which require low cost solutions rather than maximum performance (see *Figure 2: e200z0h block diagram*).
CPU is compatible with the Power Architecture VLE (Variable Length Encoding) instruction set, that allows a mix of 16-bit and 32-bit instructions, for code size footprint reduction.

The processor interface to the external memory space is based on AMBA AHB2.v6 definition, with separate 32-bit data and instruction busses.

These memory busses are connected to the system memories and peripherals through a 32-bit crossbar (XBAR). The two e200z0h processors, the DMA and RAM run up to 80 MHz.

From a software perspective, every bus master (CPU and DMA) sees every memory and peripheral (SRAM, Flash and peripherals) in a consistent, flat memory address space.

Figure 1.SPC570Sx Block diagram

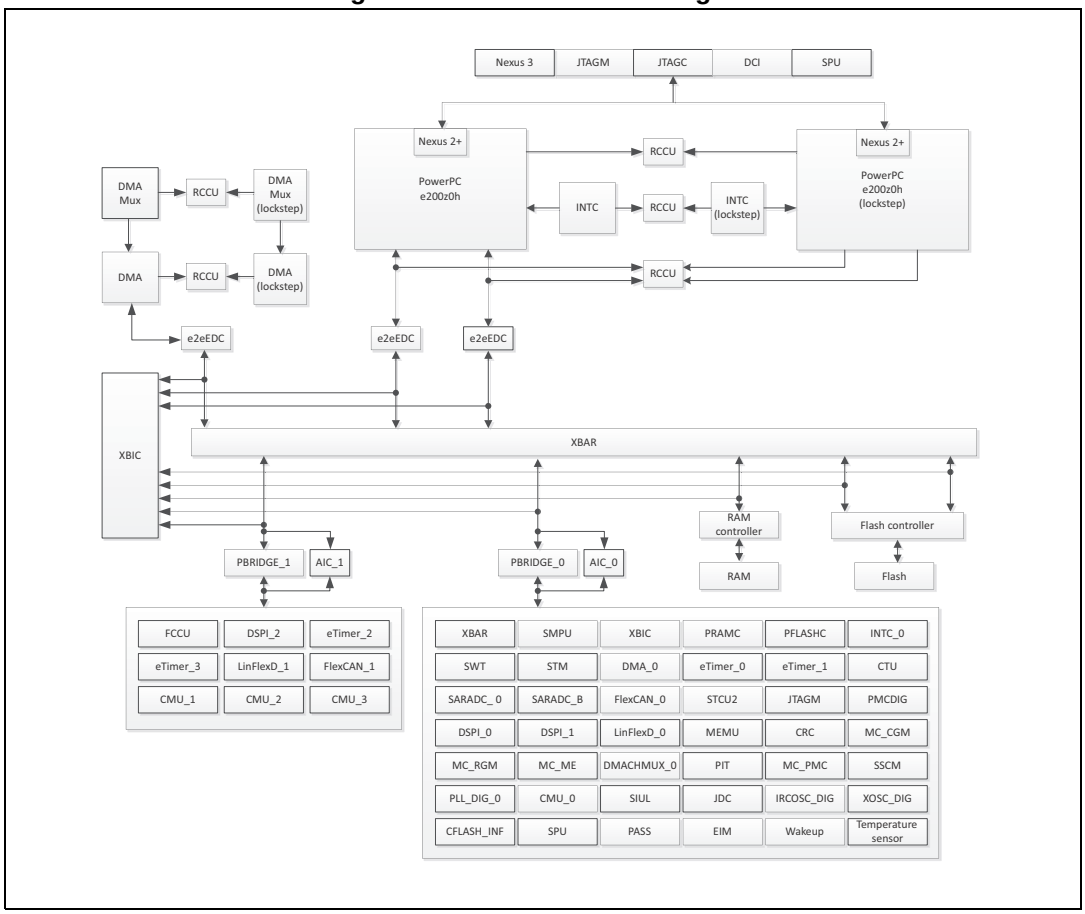
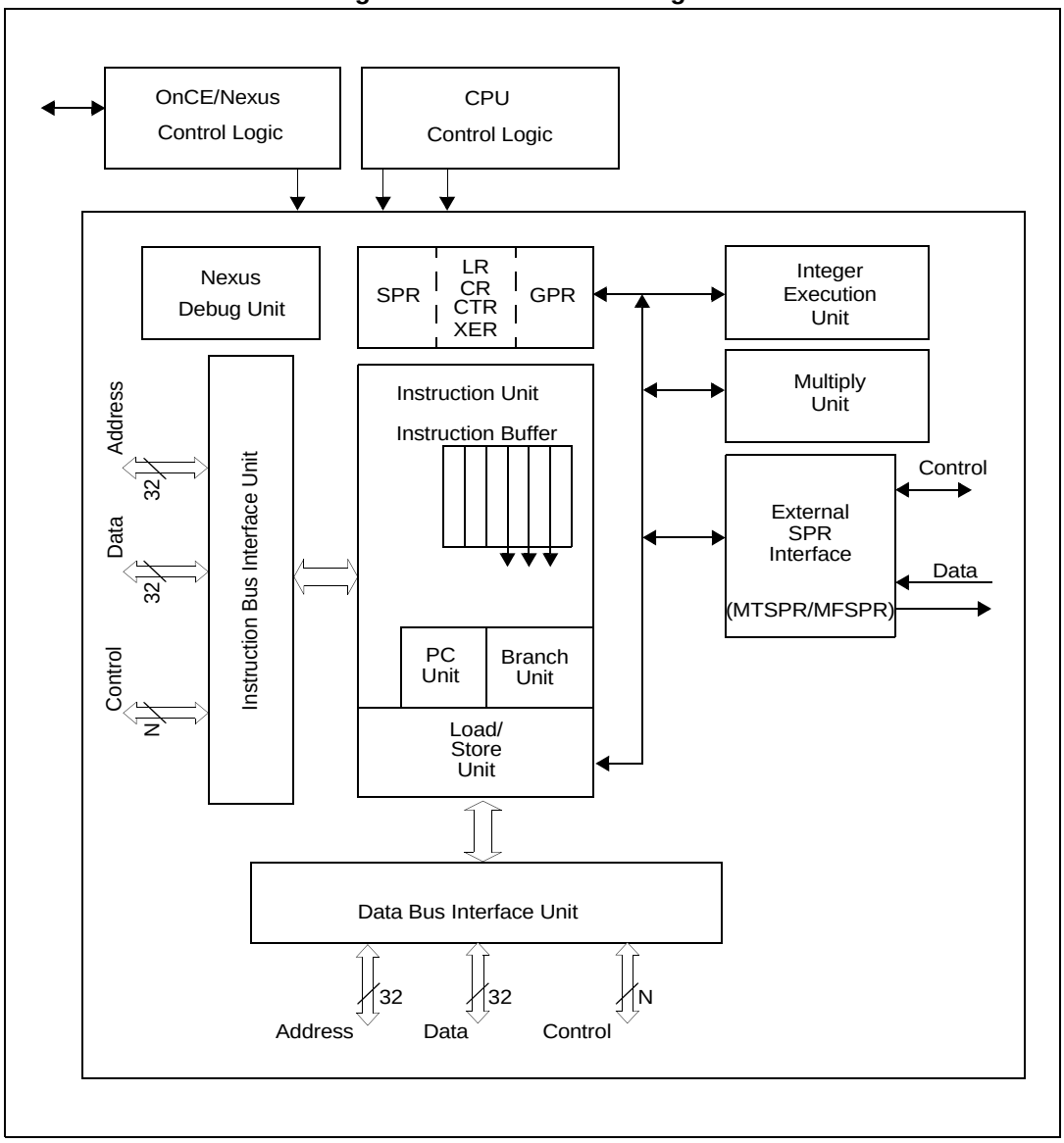


Figure 2. e200z0h block diagram



2.1 Differences across sub-family

Table 1. Sub-Family features comparison

	SPC570S40x	SPC570S50x
Flash	256KB(1)	512KB
SRAM	32KB(2)	48KB
FlexCAN	1(3)	2
Other	aligned to Superset	

1. 128KB Block 0 and 128KB Block 1 excluded on SPC570S40x.
2. [0x4000_8000 - 0x4000_BFFF] out of complete [0x4000_0000 - 0x4000_BFFF] System RAM only available on SPC570S50.
3. FlexCAN1 excluded on SPC570S40x.

Table 2. Flash Memory Map

Start Address	End address	Field	RWW partition ID
0x0040_0000	0x0040_1FFF	Flash 8 KB UTest block	1
0x0040_2000	0x0040_3FFF	Reserved	
0x0040_4000	0x0040_5FFF	Flash 8 KB BAF block	1
0x0040_6000	0x0040_7FFF	Reserved	
0x0040_8000	0x0040_BFFF	Flash 16 KB Test block(1)	1
0x0040_C000	0x007F_FFFF	Reserved	
0x0080_0000	0x0080_1FFF	Flash 8 KB Data Block 0	2
0x0080_2000	0x0080_3FFF	Flash 8 KB Data Block 1	2
0x0080_4000	0x0080_5FFF	Flash 8 KB Data Block 2	2
0x0080_6000	0x0080_7FFF	Flash 8 KB Data Block 3	2
0x0080_8000	0x00FB_FFFF	Reserved	
0x00FC_0000	0x00FC_3FFF	Flash 16 KB Code Block 0	1
0x00FC_4000	0x00FC_7FFF	Flash 16 KB Code Block 1	1

Table 2. Flash Memory Map (continued)

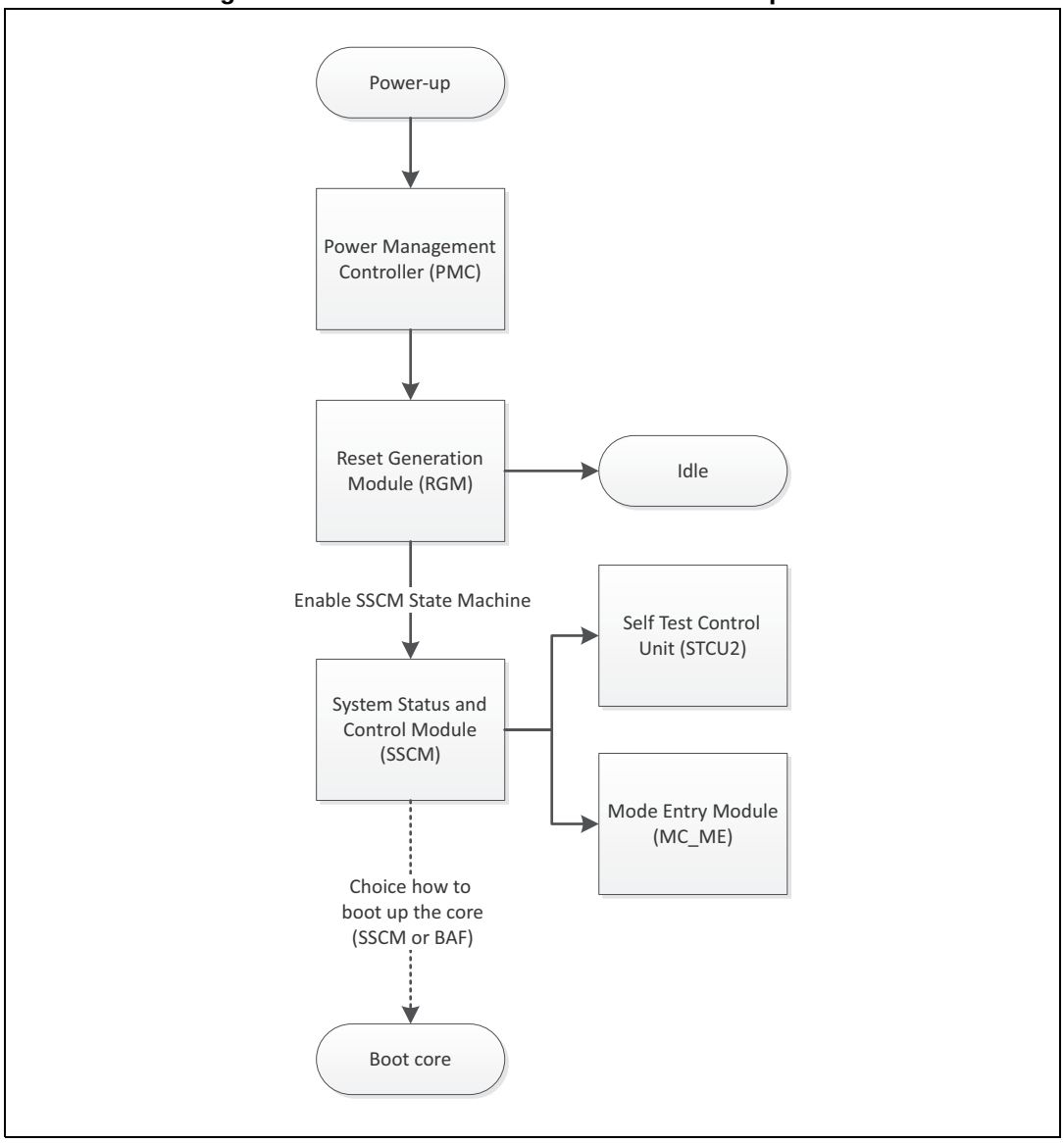
Start Address	End address	Field	RWW partition ID
0x00FC_8000	0x00FC_BFFF	Flash 16 KB Code Block 2	1
0x00FC_C000	0x00FC_FFFF	Flash 16 KB Code Block 3	1
0x00FD_0000	0x00FD_7FFF	Flash 32KB Code Block 0	1
0x00FD_8000	0x00FD_FFFF	Flash 32KB Code Block 1	1
0x00FE_0000	0x00FE_FFFF	Flash 64 KB Code Block 0	1
0x00FF_0000	0x00FF_FFFF	Flash 64 KB Code Block 1	1
0x0100_0000	0x101_FFFF	Flash 128 KB Code Block 0(2)	1
0x0102_0000	0x0103_FFFF	Flash 128 KB Code Block 1(3)	1
0x0104_0000	0x3FFF_FFFF	Reserved	

1. The Test Flash Block is not accessible directly through the Flash controller. It can be accessed only through a private bus by the SSCM module during the reset and by PASS module when a password match comparison is requested. The table reports the test Flash Block address, as it were visible from the Flash controller. In reality, the SSCM and PASS modules access the Test Flash block using the real Flash block address, which is 0x80_0000 till 0x80_BFFF.
2. Flash Block excluded on SPC570Sx40.
3. Flash Block excluded on SPC570Sx40.

3 Reset and boot

The power-up reset sequence always begins with the application of the power and follows different sequences depending on enabled option settings in the DCF records.
This allows the user to control several configuration parameters and customize the final behavior of the device (see *Section D.1: Reference documents*).
There are several modules used to step through the sequence needed to properly reset the SPC570Sx and prepare it to fetch the first instruction of the user application code (see *Figure 3: Modules involved in Reset and Boot process*).
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), the Self-Test Control Unit (STCU2) and the BAF (Boot Assist Flash) (see *Section D.1: Reference documents* for further info on these modules).

Figure 3. Modules involved in Reset and Boot process



Since this document is focused on how to boot-up the device from the internal flash (with no use of security/safety functions) in the following sections will be detailed how the user has to configure the device (and its own code) to choose the proper behavior.

3.1 Boot from internal flash

During the reset sequence, the Reset Generation Module (RGM) enables the SSCM which continues with the reset or boot-up sequence (see *Figure 4: Processor Boot*). Once the SSCM locates the boot header, which is stored in Flash (see *Table 3: Boot record structure*), it writes the start address information for the core, according to the system configuration, to special location (CADDR register) in the Mode Entry Module.

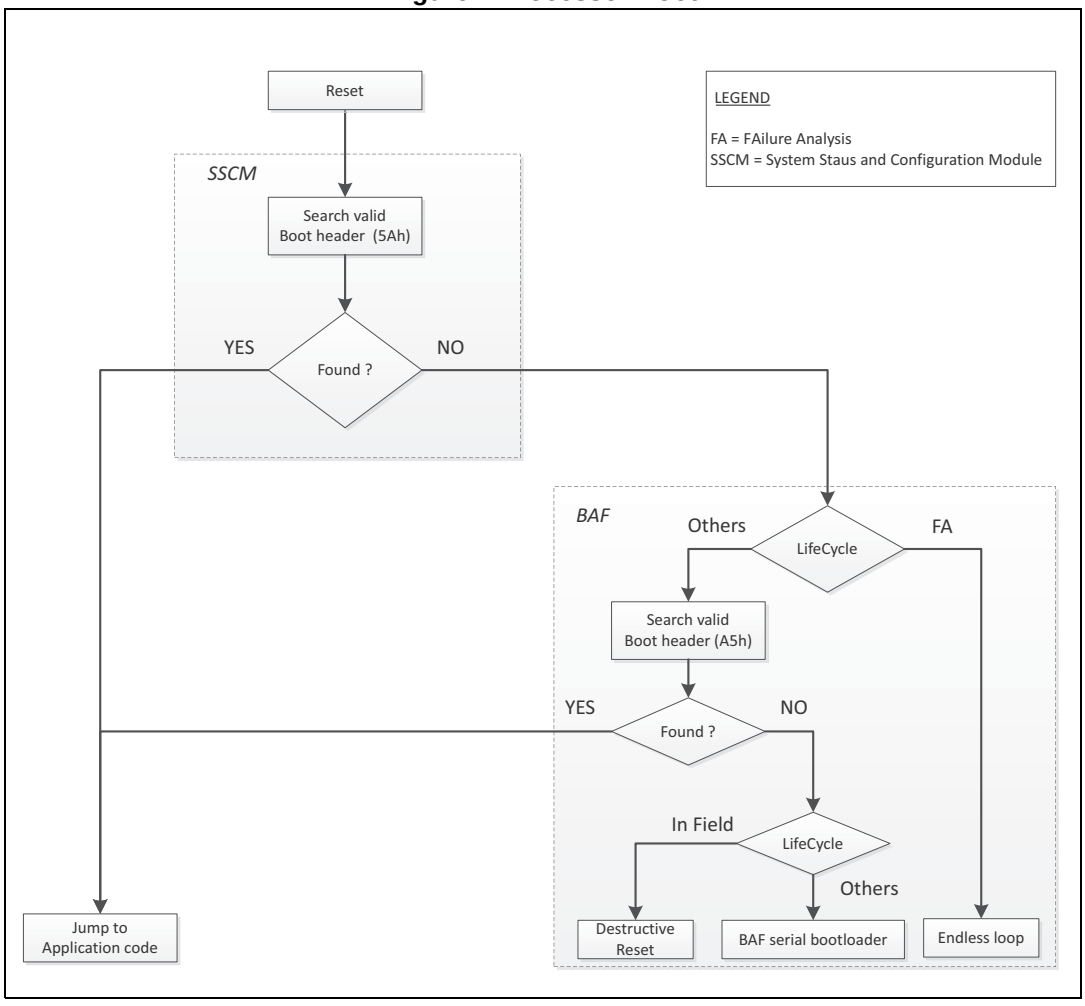
As the boot-up sequence progresses, the SSCM instructs the MC_ME to transfer the reset vector (execution starting address) to the core which starts executing.

The system configuration is set as application boot records and is programmed by the customer. It is used to set the customer application entry point after reset or after the BAF execution (if enabled).

Figure 4 shows the various possible system boots. The application entry point can be retrieved both in HW by the SSCM (see *Section 3.1.2: HW boot record search (SSCM search)*) or in SW by the BAF (see *Section 3.1.3: SW boot record search (BAF search)*).

In both cases, the application entry point is retrieved by searching for a valid Boot Record (see *Table 3: Boot record structure*), which shall be programmed by the customer in one of the pre-determined Flash memory locations (see *Table 4: Boot record search locations*) looked at by the SSCM or the BAF code.

Figure 4.Processor Boot



3.1.1 Boot Header format and Boot search locations

The boot header (see *Table 3: Boot record structure*) is used to supply a software entry point to the core: It is written by the user and loaded into specific locations in the flash memory (see *Table 4: Boot record search locations*).

Table 3. Boot record structure

Offset	Field description	
0x0	Boot Record Tag(1)	Reserved
0x4	Application start address	
0x8	Reserved	
0xC	Reserved	

1. 5Ah for HW boot record (SSCM), A5h for SW boot record (BAF)

The valid boot record for the SSCM is identified by the 5Ah tag that is different from tag used for the BAF that is identified by A5h (see *Table 3: Boot record structure*).
This will make it possible to program the same boot record locations (see *Table 4: Boot record search locations*) and launch the application directly after the reset *(see Section 3.1.2: HW boot record search (SSCM search))* or after the BAF code execution (see *Section 3.1.3: SW boot record search (BAF search)*).

The Boot header is programmed into the flash memory at the same time the user application program is programmed into flash memory.
Table 4 lists the boot search locations present in the SPC570Sx family. The “Search Order” column specifies which location is searched first.

Table 4. Boot record search locations

Search order(1)	Address	Location	SSCM configuration parameter
1	0x00FC_0000	Flash 16 KB Code Block 0	RCHW_LO_0
2	0x00FC_4000	Flash 16 KB Code Block 1	RCHW_LO_1
3	0x00FC_8000	Flash 16 KB Code Block 2	RCHW_LO_2
4	0x00FC_C000	Flash 16 KB Code Block 3	RCHW_LO_3
5	0x0100_0000	Flash 128 KB Code Block 0	RCHW_LO_4
6	0x0102_0000	Flash 128 KB Code Block 1	RCHW_LO_5
7 (Last)	0x0040_0000	Flash 8 KB BAF Block	RCHW_LO_6

1. SPC570S40x doesn’t have 128 KB Code Blocks 0 and 1.

Note: The location of the boot record no. 7 is placed within the BAF block itself, making it not erasable (the BAF block is OTP): This boot record will point to the BAF entry point.
This boot record is the last one being searched by the SSCM (see Section 3.1.2: HW boot record search (SSCM search)) and this means that, if no other boot record is programmed,

the software will start from the BAF (see Section 3.1.3: SW boot record search (BAF search)).

3.1.2 HW boot record search (SSCM search)

The SSCM module looks (see *Figure 4: Processor Boot*) for a valid software entry point called SSCM boot record (see *Table 3: Boot record structure*).
The “Application start address“ value will be stored to the CADDR register of MC_ME, which is used to drive the CPU start address after the reset phase (see *Figure 3: Modules involved in Reset and Boot process*).
Depending upon the fact that a valid SSCM boot record is found two actions can happen:

- if a valid Boot Record is programmed by the customer, then the application is executed after the reset from the specified entry point address;
- if no valid Boot Record is programmed by the customer, then the default BAF Boot Record (see *Table 4: Boot record search locations*) is retrieved and the BAF code is executed after the reset.

3.1.3 SW boot record search (BAF search)

The BAF is executed if no valid Boot Record (0x005A0000) is programmed by the customer (see *Figure 4: Processor Boot*).

Note: In this case a valid SSCM boot record pointing to the BAF code (programmed by factory at the location number 7) is found (see Table 4: Boot record search locations).

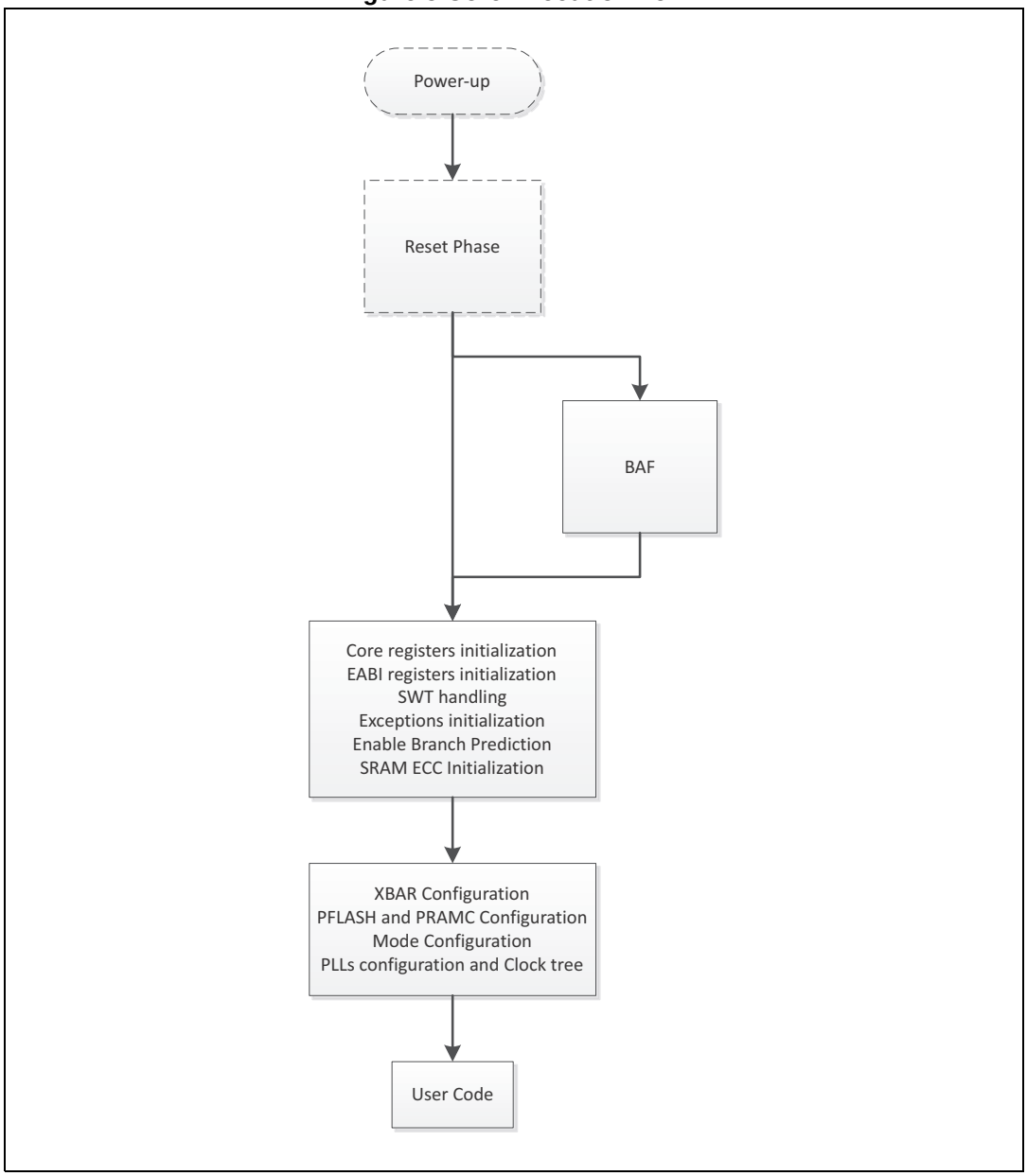
When the BAF code is executed after the reset sequence is completed, a valid application Boot Record for the application entry point is searched. The mechanism is the same executed by the SSCM: a set of fixed locations are analyzed (see *Table 4: Boot record search locations*) and, if a valid boot record (Boot record tag equal to A5h) is found, then, at the end of the BAF code, the CPU will start executing from the application entry point.

If no boot header is found in the internal flash memory and the current life cycle phase of the device is less than ”In Field” (see *Figure 4: Processor Boot*), an attempt is made to download the application by a serial protocol using the LINFlexD module or FlexCAN module (see *D.1: Reference documents*). The downloaded code is then executed by the CPU.

4 SPC570Sx initialization example

As soon the device ends reset phase, in order to run an application from the Flash memory, user code has to perform a series of basic initializations (see *Section 4.1: Initialization steps*) so to make the device able to run user application taking also into account safety related initializations (core registers, SRAM, software watchdog) so to avoid safety reactions as defined by the device's safety architecture.

Figure 5.Core Execution flow



Note: Dotted blocks added only to indicate temporal dependencies.

4.1 Initialization steps

Below the list of initialization steps the user has to do to run the application from the flash for a proper execution:

- Valid boot header (see *Section 4.2: A valid Boot Header*) setting
- Core registers initialization (see *Section 4.4: Core registers initialization*)
- SWT (Software Watchdog) handling/configuration (see *Section 4.3: Software Watchdog handling*)
- BTB (Branch Target Buffer) configuration (see *Section 4.5: Enable Branch Target Buffer*)
- SRAM (ECC) initialization (see *Section 4.6: SRAM ECC Initialization*)
- XBAR configuration (see *Section 4.8: XBAR Configuration*)
- Flash and SRAM controllers configuration (see *Section 4.9: Memory Controllers configuration*)
- Device Mode configuration (see *Section 4.10: Mode Entry Module: Configuration*)
- PLL and clock tree (selectors/dividers) configuration (see *Section 4.11: Clock and PLL configuration*)

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

4.2 A valid Boot Header

In order to boot up correctly the device (in the hypothesis of this document) the user has to provide a valid boot header (see *Section 3.1.1: Boot Header format and Boot search locations*).

Below a scratch of code to define (with GHS syntax) a valid boot header for SPC570Sx:

```
.section .rchw
#ifndef BAF
.LONG 0x005A0000     #/* Valid BH tag if SSCM is choosen */
#else
.LONG 0x00A50000     #/* Valid BH tag if BAF is choosen */
#endif
.LONG _application_start #/* Device Reset Vector (SW entry point)*/
```

4.3 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.

The SPC570Sx device family has only one SWT. Look at Reference Manual (see *D.1: Reference documents*) for more information on configuring and using the SWT.

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.

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.

In this document it is supposed 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;

Below a piece of code[(a)] 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 ;#/* load r4 with SWT base address */
 e_or2i  r4, swt_baseaddr@l

;#/* Clear soft lock */
 e_li    r3, 0xC520         ;#/* load r3 with first value */
 e_stw   r3, 0x10(r4)       ;#/* store r3 to SWT Service Register */

 e_li    r3, 0xD928         ;#/* load r3 with first value */
 e_stw   r3, 0x10(r4)       ;#/* store r3 to SWT Service Register */

 e_lis   r3, 0xFF00       ;#/* load to r3 SWT Conf . WEN = 0 */
 e_or2i  r3, 0x010A

 e_stw   r3, 0(r4)       ;#/* store r3 to SWT Control Register */
.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.

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

4.4 Core registers initialization

4.4.1 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 5: Power Architecture EABI Registers*).

Table 5. 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 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 codes, 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).
```

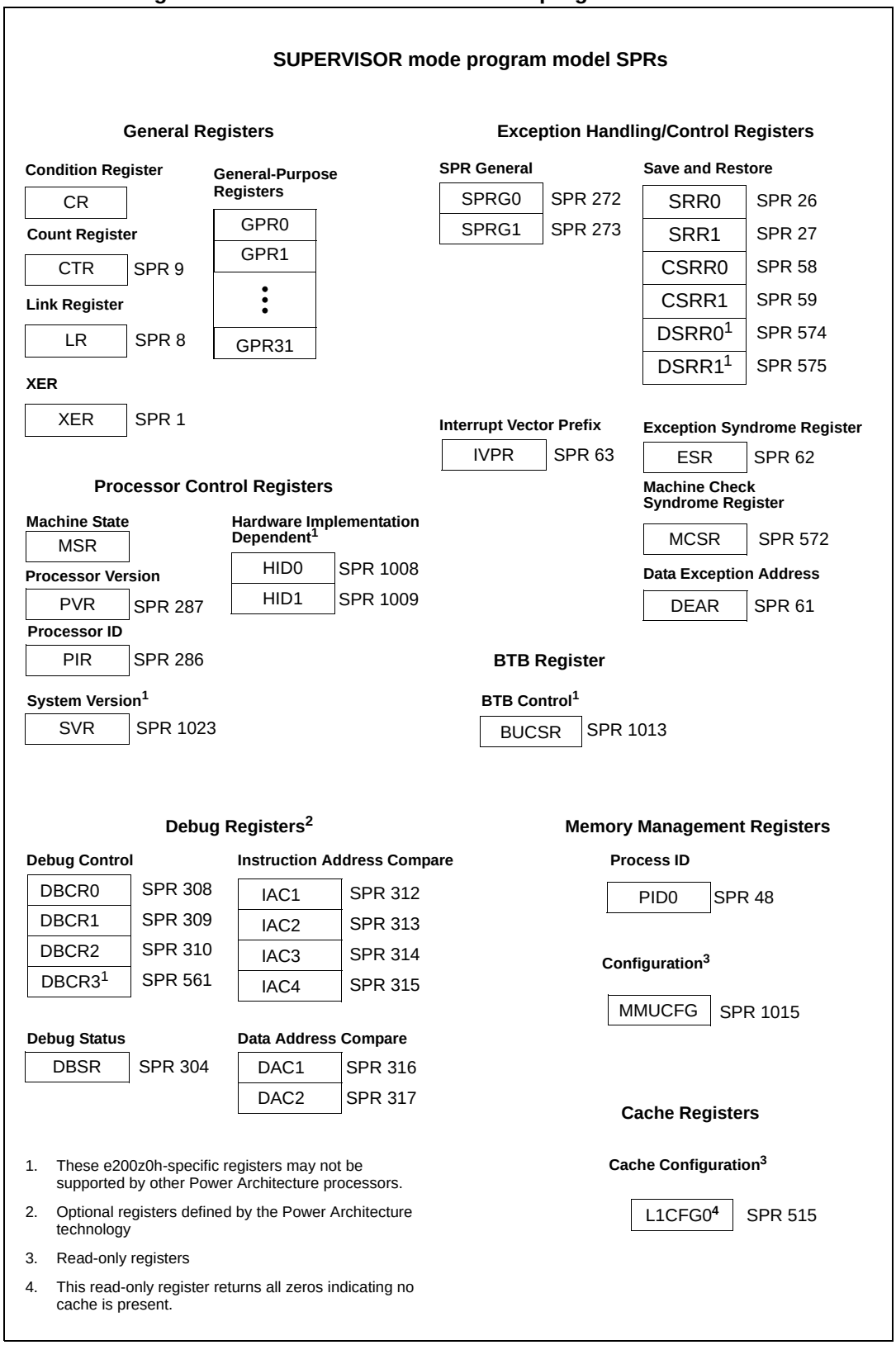
Note: This initialization step can be avoided by the user if the compiler has a startup library support (see Figure 8: Booting flow using GHS startup libraries).

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.

4.4.2 Core Register initialization

Since in the SPC570Sx family the core 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 B: Core registers initialization code*).

Figure 6. e200z0h SUPERVISOR mode program model SPRs



4.5 Enable Branch Target Buffer

The SPC570Sx 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 flush, invalidate the buffer and enable branch prediction: This can be also accomplished with a single write to the Branch Unit Control and Status Register (BUCSR) in the core.
Below, a scratch code useful to configure the BTB is listed:

```
;#----------------------------------------------------------------------#
;# Flush and Enable BTB - Set BBFI and BPEN fields (BUCSR register) #
;#----------------------------------------------------------------------#
;#/* Invalidate BTB */
e_lis r3, 0x200    ;#/* Set BBFI bit in BUCSR */
mtspr 1013, r3     ;#/* Store r3 to BUCSR special purpose register */
se_isync           ;#/* Mandatory sw synchronization for BUCSR */

;#/* Enable BTB */
se_li r3, 0x1      ;#/* Enable BTB - Set BPEN bit in BUCSR */
mtspr 1013, r3     ;#/* Store r3 to BUCSR special purpose register */
se_isync           ;#/* Mandatory sw synchronization for BUCSR */
```

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.

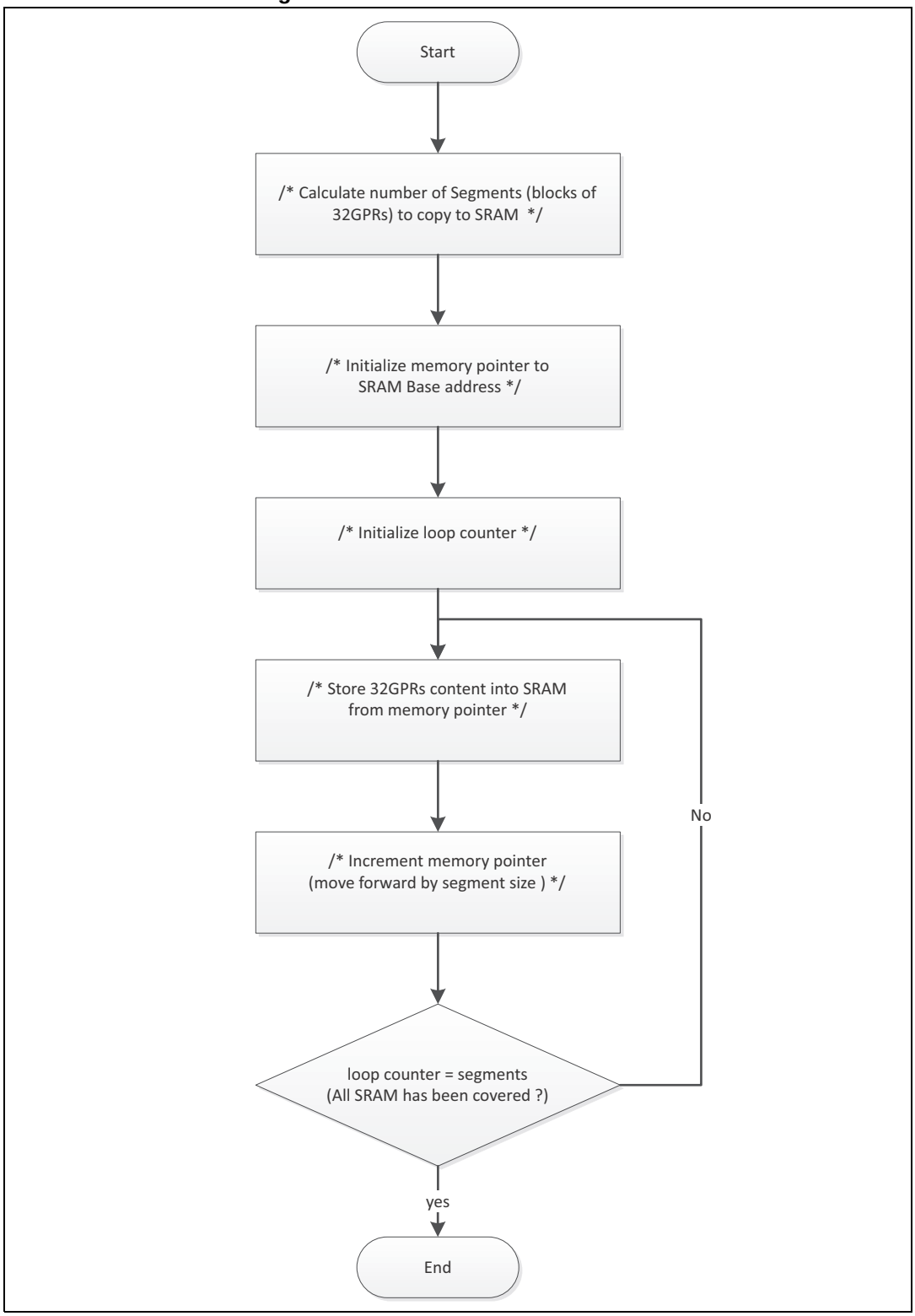
4.6 SRAM ECC Initialization

The on-chip general-purpose SRAM in the SPC570Sx family (as other SPC devices) has the ECC (Error Correction Code) protection (see *Section 2.1: Differences across sub-family* for SRAM size).

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 32-bit write operations prior to any read accesses to avoid any ECC error and therefore an exception being raised (see *Figure 7: SRAM ECC Initialization flow*).

Figure 7.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
;#/*====================================================================*/
```

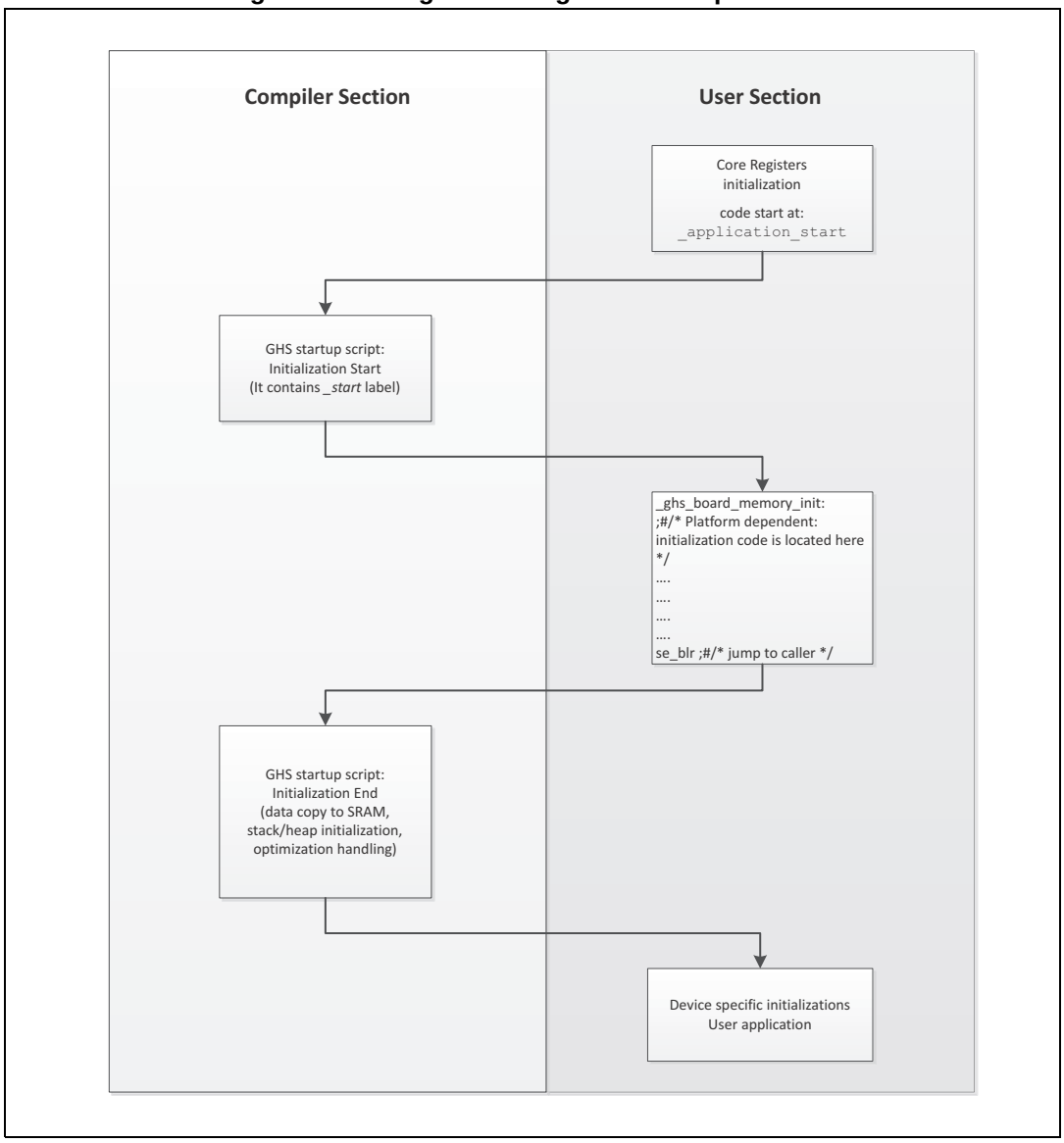
4.7 Environment Initializations

The Stack/Heap memory sections and the EABI registers, must be initialized (see *Section 4.4: Core registers initialization*) 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 from scratch by the user or left to a pre-built compiler initialization script usually available in all compiler toolchains (see *Figure 8: 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 suggested in order to facilitate the code development and description.

This means that the first instruction executed by the core after register initialization (see *Section 4.4: Core registers initialization*) is a GHS library instruction (see *Figure 8: Booting flow using GHS startup libraries*).

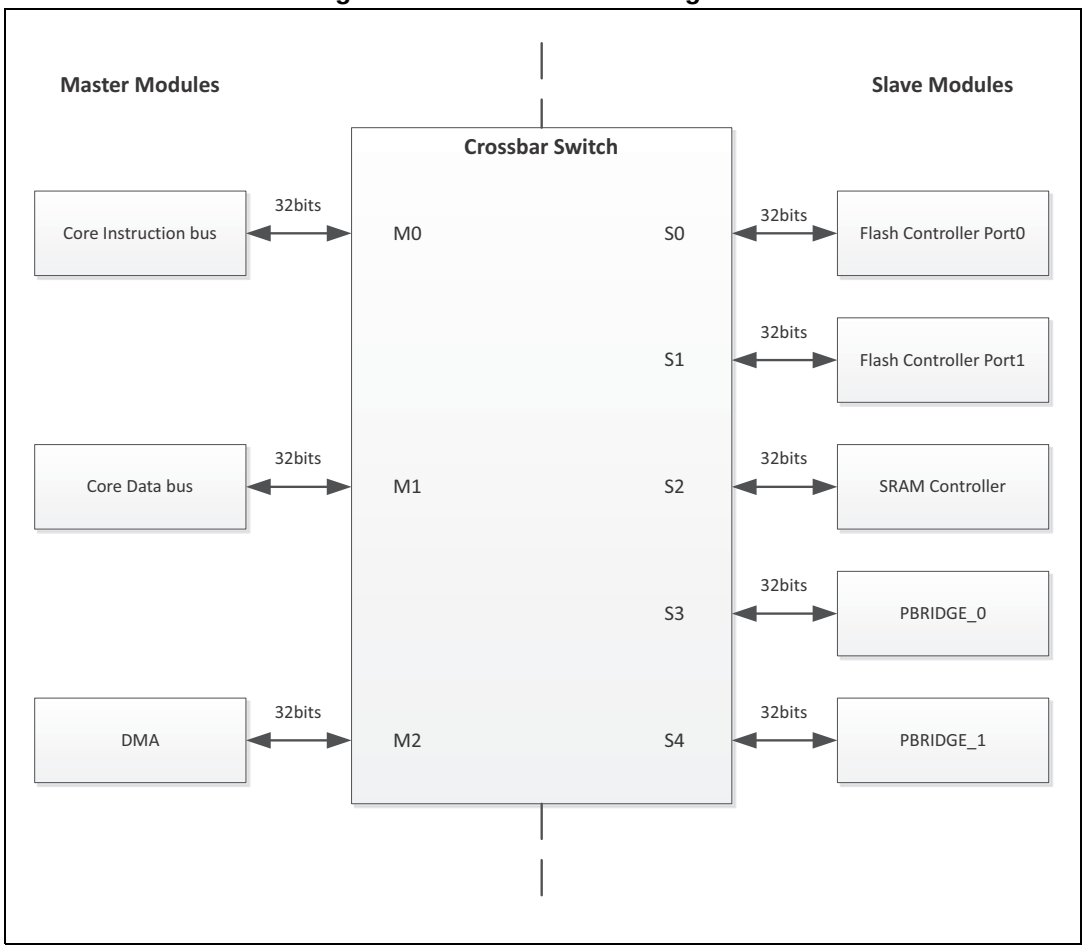
Figure 8.Booting flow using GHS startup libraries



4.8 XBAR Configuration

The crossbar switch connects bus masters and bus slaves using a hardware interconnect matrix. This structure allows all bus masters to access the different bus slaves simultaneously with no interference while providing arbitration among the bus masters when they access the same slave. A variety of bus arbitration methods and attributes may be programmed on a slave-by-slave basis.
This devices family contains only one (XBAR_0) crossbar switch module.

Figure 9.Crossbar switch integration



It is suggested to configure XBAR in order to balance data workloads across the architecture and to improve application performances.

For this purpose a control register is available (XBAR_CRSn) for each slave (see *Section 4.8.1: XBAR Registers configuration*).

4.8.1 XBAR Registers configuration

The master priority assignments for the masters on a per-slave-port basis (readable through XBARn Priority Registers) are hardwired (0x0000_0123h for PRSn registers).

The master policy during undefined length bursts for the master are configurable through Master General Purpose Control Registers (XBAR_MGPCRn).

The slave behavior is configurable through the Slave control registers (XBARx_CRSn) where user can choose arbitration policy and parking control strategy.

Below a piece of code that shows an example on how to configure the XBAR: in particular slaves no. 0,2,3,4 are parked on data bus (Master port no.1) while slave port no. 1 is parked on master port no. 0 (see also *Table 6: Flash memory port assignment*).

```
/* FLASH Controller Port0 */
(* (uint32_t *)0xFC004010) = 0x00FF0001;          /* S0 Slave: DBUS(M1) */
/* FLASH Controller Port1 */
```

```
(* (uint32_t *)0xFC004110) = 0x00FF0000;          /* S1 Slave: IBUS(M0) */

/* System RAM */
(* (uint32_t *)0xFC004210) = 0x00FF0001;          /* S2 Slave: DBUS(M1) */

/* PBRIDGE_0 */
(* (uint32_t *)0xFC004310) = 0x00FF0001;          /* S3 Slave: DBUS(M1) */

/* PBRIDGE_1 */
(* (uint32_t *)0xFC004410) = 0x00FF0001;          /* S4 Slave: DBUS(M1) */
```

4.8.2 Flash Memory Access

The access to the flash memory via slave ports 0 and 1 is routed based on the master number so that the routing has been chosen to balance the data traffic in order to facilitate the core instruction fetches. *Table 6* shows the flash memory port assignments.

Table 6. Flash memory port assignment

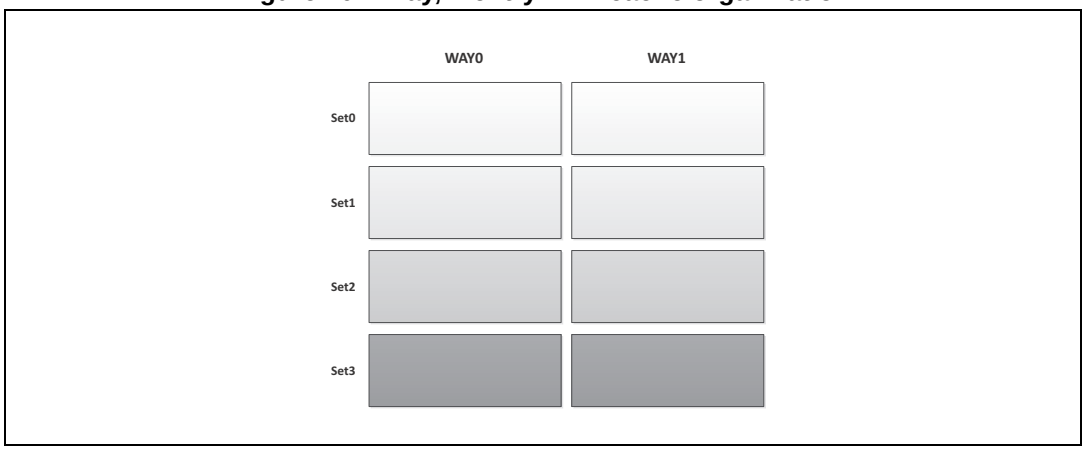
Flash memory slave port 0	Flash memory slave port 1
e200z0 CPU DBUS (M1)	e200z0 CPU IBUS (M0)
DMA Master (M2)	--

4.9 Memory Controllers configuration

4.9.1 Flash Controller configuration

The flash memory controller supports 2 32-bit AHB buses and a 128-bit read data interface to the flash memory array (see *D.1: Reference documents* for further details). Each AHB port contains a 4-entry, 2-way set-associative mini-cache (see *Figure 10: 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 *D.1: Reference documents*).
Each mini-cache entry is 128 bits in size, matching the code flash array page size and providing 128 bytes of high-speed local storage.

Figure 10.2-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/PFCR2 registers (PFCR stands for Platform Flash Configuration Register).
The arbitration of concurrent flash access requests from the two AHB ports of the flash memory controller and allocation[(b)] of the line read buffers are controlled via the PFCR3 control register (Platform Flash Configuration Register 3).

Note: The user can control 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 (see D.1: Reference documents).

Parameters affecting the transfers between the flash array and the core, such as read wait states, are not optimal out of reset and to obtain the best performances it should be

b. 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.

configured for desired target system clock frequency *(see D.1: Reference documents)*.

Note: When modifying characteristics for a memory, 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 registers will be executed from the SRAM.

4.9.2 SRAM wait State

The user, through the configuration of PFCR1 (Platform RAM configuration Register 1), can specify operation of the RAM controller[c] (see *D.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 32-bit read burst for the port. 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.

4.10 Mode Entry Module: Configuration

The SPC570Sx device family (as other SPC56xx devices) has several operating modes (see *D.1: Reference documents*).

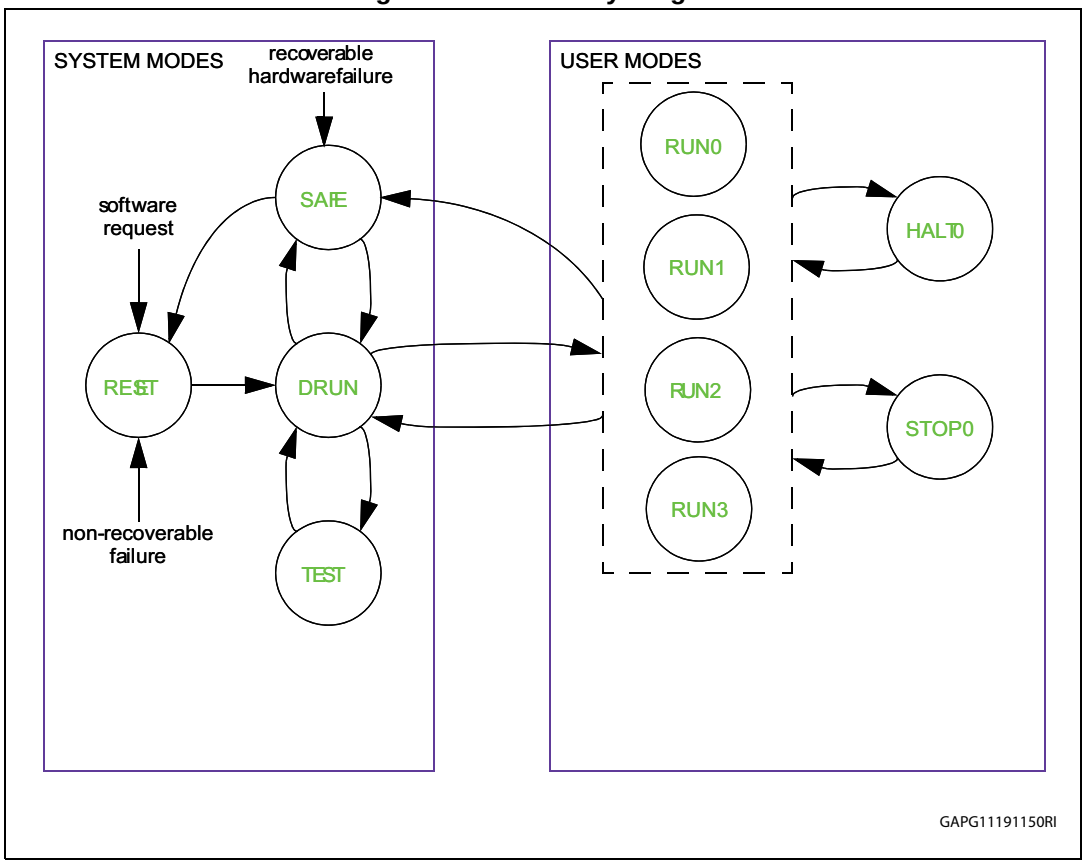
The MC_ME controls the chip mode and mode transition sequences in all functional states (see *Figure 11: Mode Entry Diagram*). It also contains the configuration, control and status registers accessible for the application.

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.

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

c. The RAM controller supports two 32-bit AHB interfaces and a 32-bit RAM array interface. The primary AHB port provides a connection to the platform crossbar for direct RAM access from the various crossbar masters.

Figure 11.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 */
```

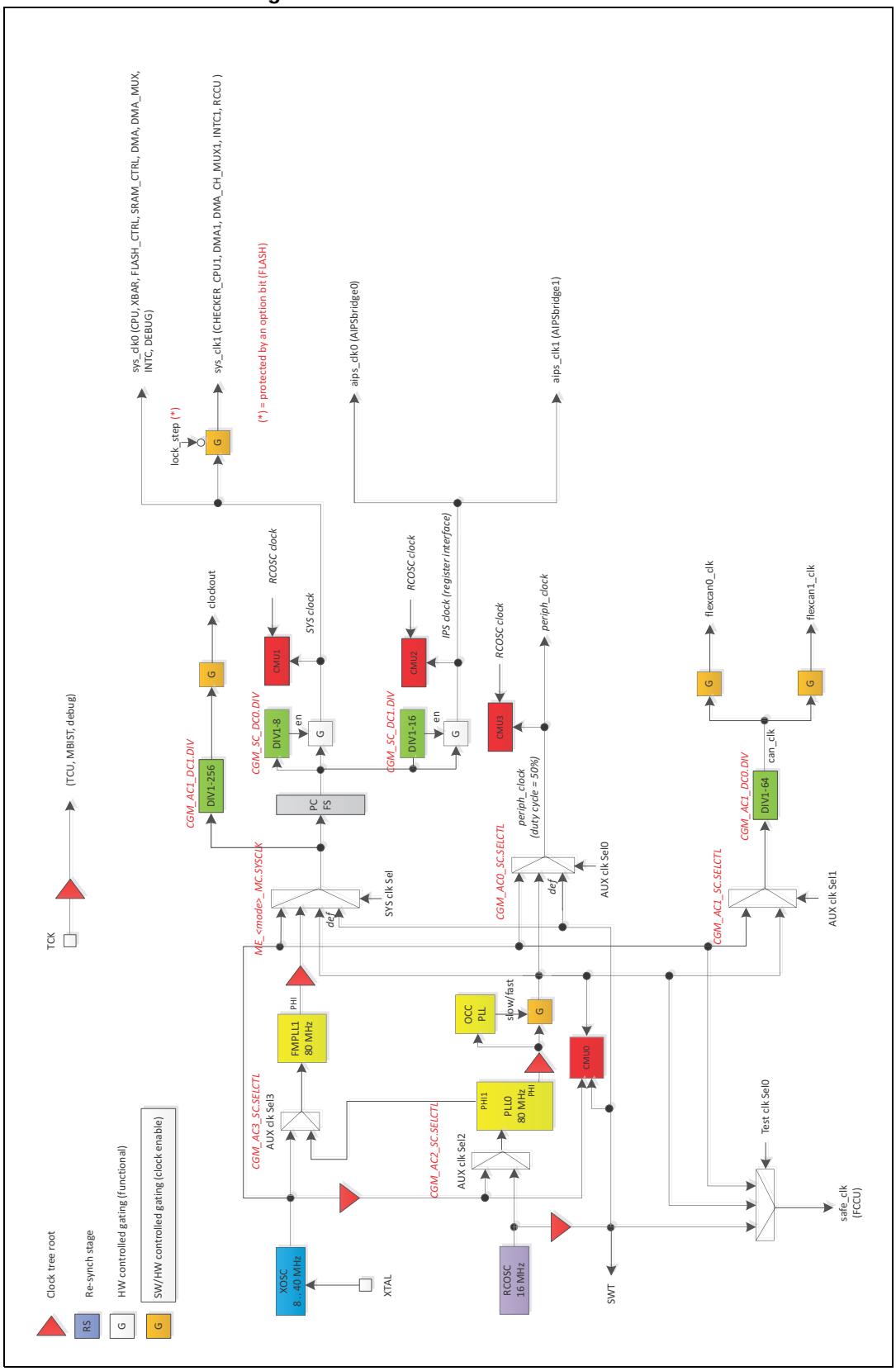
```
/* Wait for mode entry to complete */
while(MC_ME.GS.B.S_MTRANS == 1);
/* Check DRUN mode has been entered */
while(MC_ME.GS.B.S_CURRENT_MODE != # DEFINE DRUN_MODE 0x3); /
```

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 D.1: Reference documents).

4.11 Clock and PLL configuration

The device has 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 core and each IP inside the devices to run up to the desired frequency (see also System clock frequency limitations chapter listed into *D.1: Reference documents*).
The system clock tree (CLKOUT belongs to this sub-tree) (see *Figure 12: SPC570Sx Clock Generation*) which includes CPU (as well as all modules that run in the platform domain), memories and debug logic is independent from the peripheral clocks.

Figure 12.SPC570Sx 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. See also “Mode Entry Module (MC_ME)” chapter listed into *D.1: Reference documents*.

There are three possible ways to provide the source clock:

- External oscillator
- External crystal
- Internal 16 MHz RC oscillator (IRCOSC)

From one of these input sources, the internal clocks can also be generated from one of the two PLLs (PLL0[d] and PLL1[e]), 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).
Each of the outputs of the module clock selectors has individual dividers that allow for more clock frequency granularity for a given peripheral (see *Appendix C: Clocks and PLLs Initialization example* for reference).

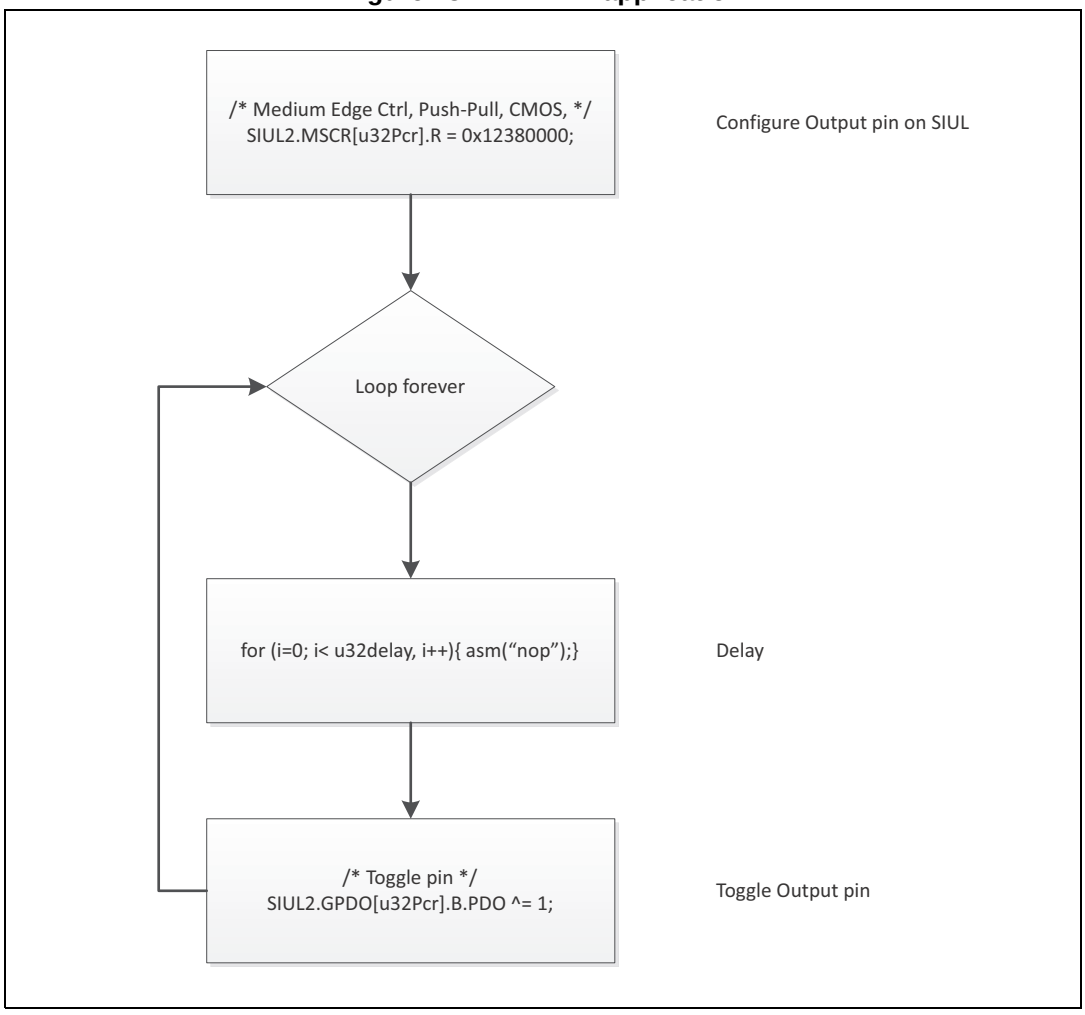
d. The PHI1 output of PLL0 can also be used as the clock source for PLL1.

e. PLL1 can drive only SYS clock tree

5 Blink LED application

As soon the core has ended its own initialization flow (see *Section 4: SPC570Sx initialization example*) the user can finally execute his application. In the example described in this document a blink led code is shown (see *Figure 13: Blink LED application*).
After the pin configuration through MSCR register inside SIUL2 peripheral (see *D.1: Reference documents*) the application loops forever changing output state to the pin configured by the user.

Figure 13.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 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:
```

Note: __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 Core 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 SRR0, r1
 mtspr SRR1, r1
 mtspr CSRR0, r1
 mtspr CSRR1, r1
 mtspr DSRR0, r1
 mtspr DSRR1, r1
 mtspr DEAR, r1
 mtspr IVPR, r1
 mtspr 62, r1       ;#ESR
 mtspr 8,r31 ;#LR
.endm
```

Appendix C 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.AC2_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);

PLL0_Init(pll0_clk);/* Configure PLL0(Init of FMPLL configuration regs) */

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_DC1.B.DIV = DIVIDEBY2;/* Peripheral Clock Divide by 2 */
MC_CGM.SC_DC0.B.DIV = DIVIDEBY1;/* System 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_DC0.B.DE = 1; /* Enabled (already enabled after reset) */
MC_CGM.AC0_DC0.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 D Further Information

D.1 Reference documents

1. *32-bit Power Architecture® microcontroller for automotive ASILD Chassis and Safety applications* (RM0349, DocID024507)
2. *32-bit Power Architecture® microcontroller for automotive ASILD Chassis and Safety applications* (Datasheet, DocID024492)
3. Hw recommendations for SPC570Sx device family (AN4721, DocID027988)
4. Safety Manual for SPC570S family (AN4247, DocID024209)
5. SPC570Sx Errata sheet (ES0278, DocID026793)
6. Transition from SPC560P50x to SPC570S50x (AN4405, DocID025566)

D.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

Table 7. Acronyms and abbreviations (continued)

Terms	Meaning
MPU	Memory Protection Unit
PMC	Power Management Controller
PMU	Power Management Unit
PLL	Phase Locked Loop
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
07-Feb-2017	1	Initial release.