AN4253
Application note

SPC564Lx device startup flow

Introduction

This document describes the startup process of the SPC564Lx between the reset to the first instructions of the application startup code.
It describes all the necessary steps involved to successfully achieve application startup. It also describes possible modifications in the low level startup blocks that may be changed, replaced or edited depending on the system needs.

This document is intended as a guideline for software developers responsible for low level startup code and building blocks.

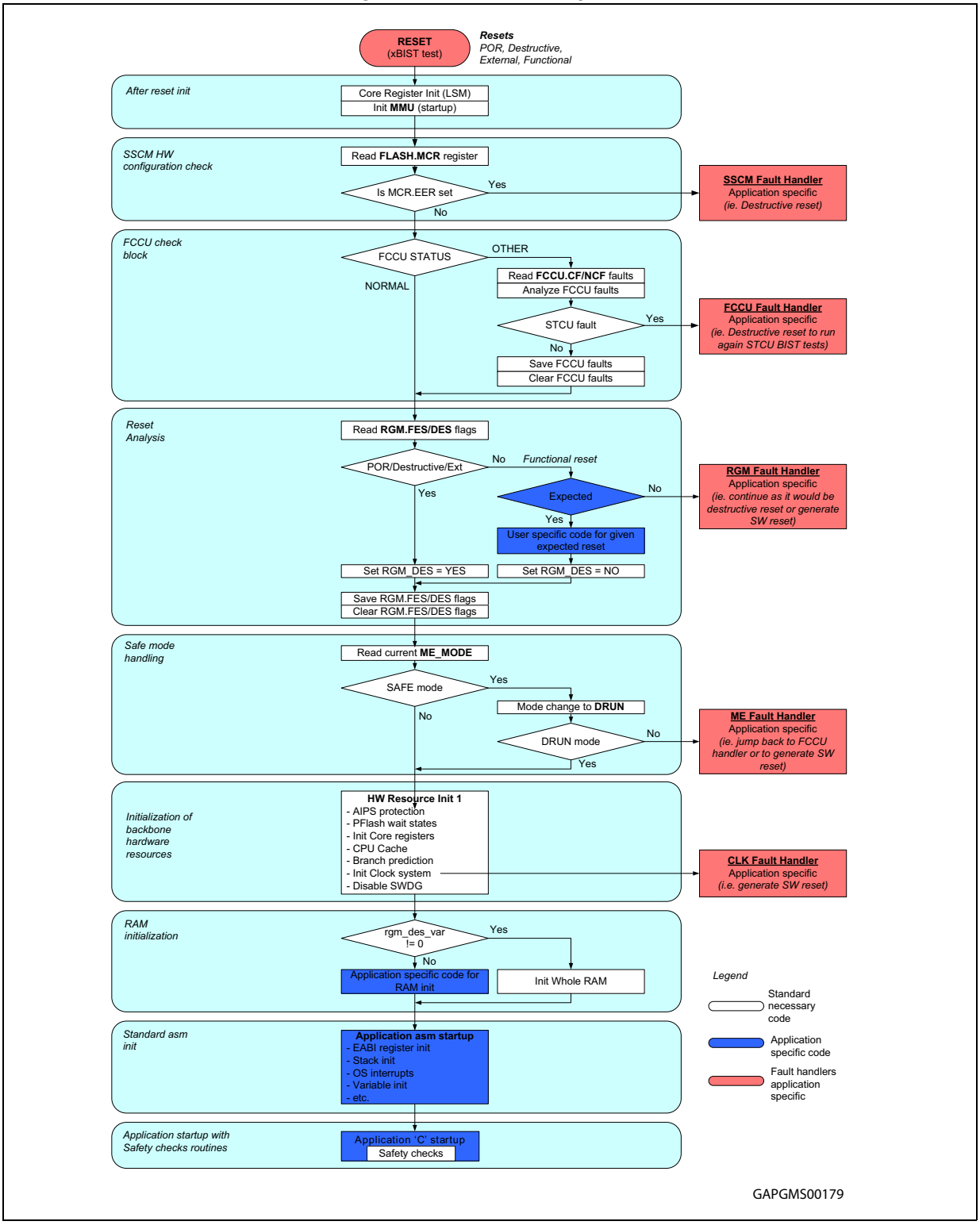
Contents

List of tables

List of figures

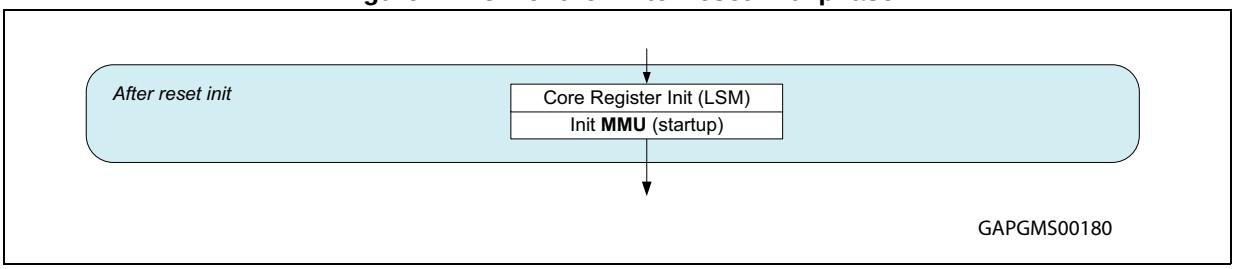
1 Startup flow diagram

Figure 1. General startup flow



2 After reset init

Figure 2. Flow of the "After reset init" phase



2.1 Core Register Init (LSM)

In Lock Step (LSM) Mode it is mandatory to initialize the core specific purpose register to avoid redundant checker faults under specific circumstances. The reason for the faults comes from the fact that not all core specific purpose registers are initialized after reset and their difference can cause RCCU fault later on.

It is not necessary to initialize the core SPRs registers when the device is operating in decoupled mode.

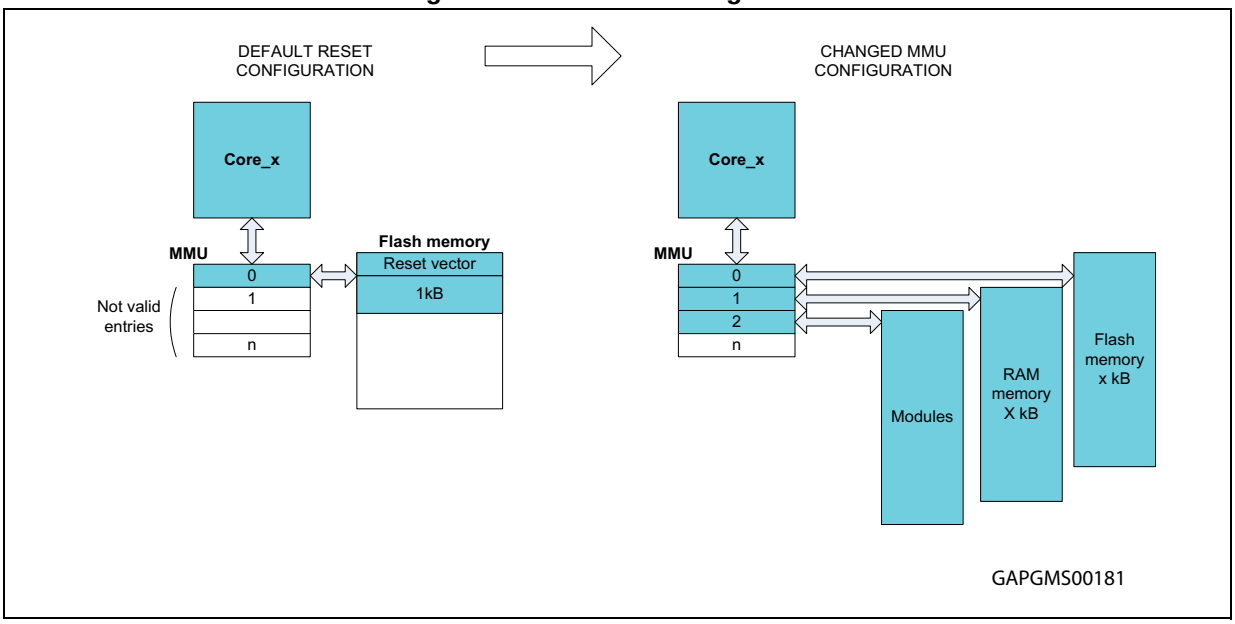
2.1.1 Code sequence

1. `Write to selected Core Specific Purpose Registers (SPRs) any value`

2.2 Memory Management Unit (MMU)

Default reset MMU configuration in each core enables instruction fetch from 4 KB memory block area starting from a reset vector address. 4 KB code area may or may not be sufficient for the startup code, but a main issue of the default MMU configuration is missing access setup for RAM memory and for peripheral modules. They must be explicitly added to the MMU table before any access is executed to them by the startup code.

Figure 3. MMU reset configuration



2.2.1 Default reset configuration

- 4 KB code Flash area starting at reset vector

2.2.2 Code sequence

Memory management unit is a part of the core with its dedicated register interface (MASx) to access an internal MMU configuration.

Read operation sequence

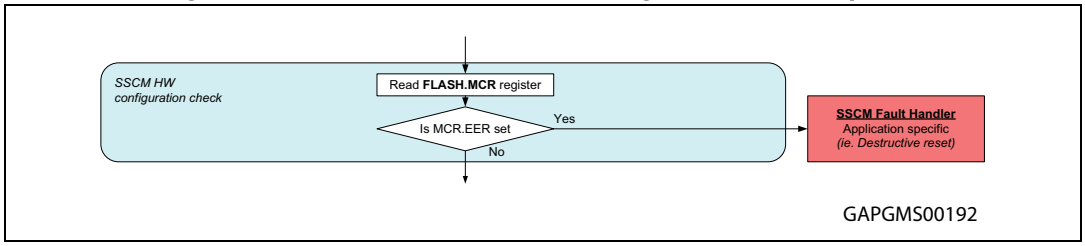
1. Prepare MMU item number in MAS0 register (**mtspr** instruction)
2. Execute **tlbre** instruction
3. Content of table entry is read to MAS0-3 register (**mfspr** instruction)

Write operation sequence

1. Prepare configuration in MAS0-3 registers (**mtspr** instruction)
2. Execute **tlbwe** instruction
3. Content of MAS0-3 registers is written to MMU entry

3 SSCM HW configuration check

Figure 4. Flow of the "SSCM HW configuration check" phase



Before the microcontroller exits the Reset phase several operations are executed. One of them is device HW configuration controlled by System Status and Configuration module (SSCM). Target of the configuration process is to setup all important analog modules to predefined factory setting plus to initialize default values of several key registers.

Configuration values are stored in the shadow flash memory region which is read by SSCM module during reset phase. As any other flash memory it can happen that values stored there are corrupted with multi bit error. Unlike the application exception system triggering ECC fault in such case, the SSCM module does not have such possibility. The system is still under reset. The SSCM module behavior is that it ignores any ECC faults that might happen during read operation.

The consequence of such silent behavior is that the device can be configured wrongly. Such faulty configuration can remain unnoticed or can have severe safety impact in the application. The distinction between the two is difficult because there is no trace information which data were corrupted and how much.

There is only one possible check to identify if an ECC fault occurred or not. Any read operation from flash memory having ECC fault leaves MCR.EER bit inside the flash module registers set regardless if the read operation is executed by the Core or SSCM module. This can be used to identify if the SSCM module has faced ECC error during configuration or not.

3.1 Code sequence

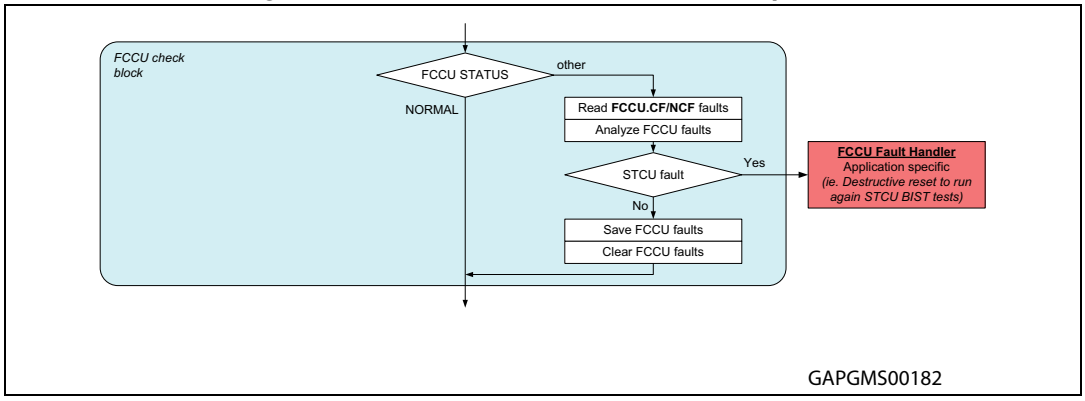
1. `Read FLASH.MCR register (0xC3F8_8000)`
2. `Test EER bit (bit 16)`
3. `If set ('1') => ECC fault has happened, might be wrong device configuration => SSCM Faul Handler`

3.2 SSCM fault handler

SSCM fault handler is an application specific piece of code. Most probable solution would be to trigger a long reset or external one to re-run SSCM configuration.

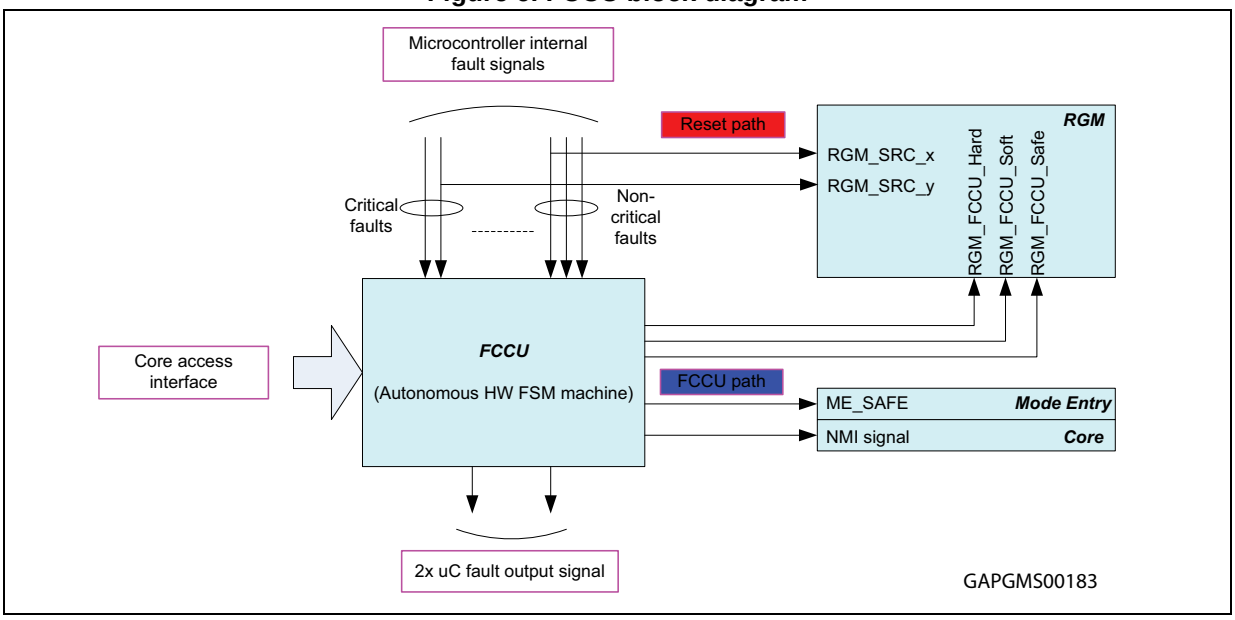
4 FCCU block

Figure 5. Flow of the "FCCU check block" phase



FCCU blocks monitors various fault signals coming from different parts of the SPC564Lx microcontroller during runtime. FCCU module works in parallel to RGM module. It provides safety measures to signal the fault when reset module fails. Details of FCCU module can be found in AN3235 (see *B.1: Reference documents*).

Figure 6. FCCU block diagram



After the reset FCCU module can be in two states depending on the reset type:

- Destructive/external resets FCCU initialized completely, all faults cleared by HW
- Functional resets FCCU untouched, status registers contain captured faults

There are three different error paths reported by FCCU when any fault occurs:

- Reset — Critical fault or non-critical fault with a configured reset reaction
- Safe mode — FCCU fault state entry always initiates safe mode request, where the HW resources are limited and the user has to solve the faults before asking for a change of the mode.
- NMI — FCCU fault state entry always generates non-maskable signal. This signal will generate an exception to the core for faults without reset generation reaction or in case when the reset is not present because of a fault in the reset module (safety measure against failures in RGM module)

Caution: Whenever microcontroller is in safe mode, all faults in the FCCU status registers must be cleared before any attempt to leave the safe mode.

4.1 FCCU status check

The first part of FCCU handler checks the state of the FCCU module after the reset. There are four possibilities:

- NORMAL — Everything is fine from the point of FCCU module, a reset was either destructive or caused by a non-critical fault that is masked in the configuration, i.e. default behavior of SW functional reset.
- ALARM — There is a non-critical fault pending in alarm state waiting for SW intervention.
- FAULT — All critical faults and non-critical faults without alarm timeout enabled.
- OTHER — Unknown state. It should not happen at all.

The FCCU handler shall resolve all states except NORMAL with the read, analyze, save and clear procedure. No accesses to FCCU registers other than reading the FCCU.CTRL.OPS shall be done between the setting of the key and the actual clear of the status flag.

4.1.1 Code sequence

```
1. Write command OPR=3 to FCCU_CTRL register
2. Wait for operation to finish (FCCU.CTRL.OPS = SUCCESS)
3. Read the FCCU_STATUS register
4. Decide if Status == 0x0 (NORMAL)
```

4.2 Read critical and non-critical faults in FCCU (FCCU.CF/NCF)

The FCCU module contains two groups of faults, critical faults and non-critical faults. Both groups should be read for later analysis.

4.2.1 Code sequence

Reading Critical Faults (CF)

1. Write command OPR=9 to FCCU_CTRL register
2. Wait for operation to finish (FCCU.CTRL.OPS = SUCCESS)
3. Read the FCCU_CF register

Reading Non-Critical Faults (NCF)

1. Write command OPR=9 to FCCU_CTRL register
2. Wait for operation to finish (FCCU.CTRL.OPS = SUCCESS)
3. Read the FCCU_NCF register

4.3 Analyze FCCU faults

In most cases this block will be skipped unless application requires special startup code for different faults.

4.4 STCU fault analysis

This analysis is recommended as any issue found during the built-in self test can signal a presence of latent faults.

There are three signals routed from STCU to FCCU unit to report different kind of issues:

- CF[20] Critical fault found by xBIST test
- CF[27] STCU fault of starting the xBIST tests during the runtime, which leads to content corruption in the RAM array or in the digital logic
- NCF[12] Non-critical fault found by xBIST test

It is recommended to monitor all three fault signals and take an appropriate action when any STCU fault is active.

4.4.1 Code sequence

1. Test bits b'20 and b'27 in Critical faults
2. Test bit b'12 in Non-critical faults
3. If any of the testing bits is '1' > Jump to FCCU Fault handler
   Otherwise continue in startup

4.5 Save FCCU faults

It is usually required to keep fault values for later analysis by the application code. This cannot be done by simply leaving status registers untouched and read them later. The reason is that any present fault in FCCU status register would prevent successful transition from SAFE mode to DRUN mode.

In such case it is necessary to store the read values somewhere else.

Some possibilities are:

- Non-volatile RAM used to exchange data between the startup code and the application. In case of destructive reset, code has to initialize non-volatile block to default values.
- One of the core registers like SPRGx. These registers are usually used by the application or operating system for non specific purposes.

4.6 Reset FCCU faults

Any fault latched by the FCCU prevents successful transition from Safe mode to DRUN. Therefore whenever the microcontroller is in safe mode, FCCU flags must be cleared before executing mode transition.

4.6.1 Code sequence

Clearing Critical Faults (CF) sequence

1. `Write the Key 0x618B7A50 to FCCU_CFK register`
2. `Write CF status value back to FCCU_CF register`
3. `Wait for operation to finish (OPS == SUCCESS)`

Clearing Non-Critical Faults (NCF) sequence

1. `Write the Key 0xAB3498FE to FCCU_NCFK register`
2. `Write NCF status back to FCCU_NCF register`
3. `Wait for operation to finish (OPS == SUCCESS)`

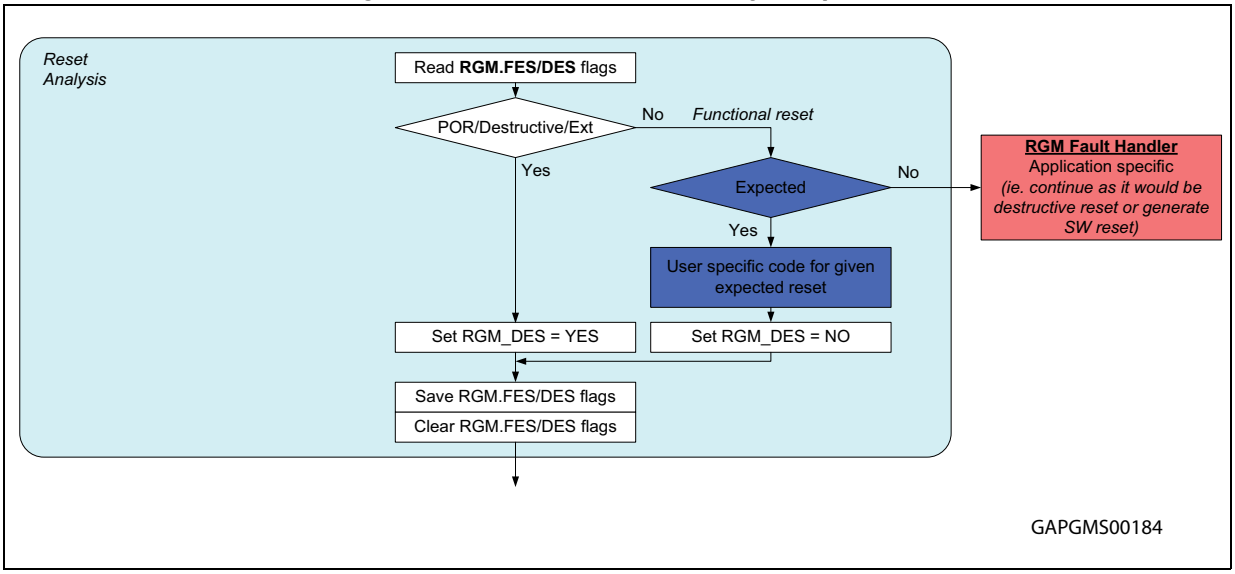
4.7 FCCU fault handler

FCCU fault handler is an application specific piece of code. There might be several possibilities like:

- Analyzing which LBIST partition or MBIST block failed and decide if it is critical for the application or not.
  Example can be failing memory block belonging to Flexray module. If Flexray module is not used then the fault found by STCU self test is of no harm for the application and startup can continue.
- Generate destructive reset to restart microcontroller from the very beginning including STCU Self tests. If the fault was because of a temporary failure, re-initiated self test will finish ok, which enables to continue with the startup.

5 RGM block

Figure 7. Flow of the "Reset analysis" phase



The reset module (RGM) centralizes the reset functionality of the microcontroller. Its decision for the reset event is based on evaluation of selected number of HW signals coming from various parts of the microcontroller.

RGM categorizes signals into two groups:

- Destructive reset signals — Signals notifying problem with the power supply with the consequence of corrupted RAM memory content. In such case complete RAM initialization is required.
- Functional reset signals — All other signals that might report serious problem in the hardware with consequence on application behavior. RGM offers limited configurability for some of them.

Complete list of reset signals is in RM0032 (see *B.1: Reference documents*).

5.1 Read RGM status (both destructive and functional resets)

Both groups of flags can be read from RGM base address with simple 32 bit read instruction. The upper 16 bits store functional reset signals, the lower 16 bits destructive ones.

5.1.1 Code sequence

1. 32b read from RGM address 0xC3FE4000

5.2 POR/destructive/external reset check

POR, destructive and external resets are destructive to RAM memory content. It means that the entire RAM memory must be initialized in the startup.

External reset is destructive only if STCU BIST tests are enabled. If STCU built–in self test is not used, external reset can be omitted from the check.

5.2.1 Code sequence

1. Do AND operation between read FES_DES value and mask 0x8000FFFF
2. If the result != 0x0 > destructive reset, RAM corrupted

5.3 Expected functional reset

This analysis is application specific as some reset can be used from the application to change the mode of the ECU.

In most cases the startup analysis requires additional information from the application to decide if the reset was expected or unexpected. One possibility is to use a small RAM memory block that serves as non-volatile storage, valid only if the reset is functional one.

5.4 User specific code for given expected reset

If the reset was expected the user provides the necessary startup code. In most case it can be by setting special flags or to set appropriate values in the non-volatile RAM block.

5.5 Save reset flags

Similar to FCCU save operation:

- Non-volatile RAM is used to exchange data between the startup code and the application. In case of a destructive reset the entire memory will be cleared.
- One of the core registers like SPRGx. These registers are usually used by the application or operating system for non specific purposes.

5.6 Clear reset flags

Reset flags are cleared with write-‘1’-to-clear operation to RGM base address (FES+DES registers).A clear operation is needed in case the microcontroller is in safe state to enable mode entry transition.

5.6.1 Code sequence

```
1.  Write '1's to RGM base address 0xC3FE4000
```

5.7 RGM fault handler

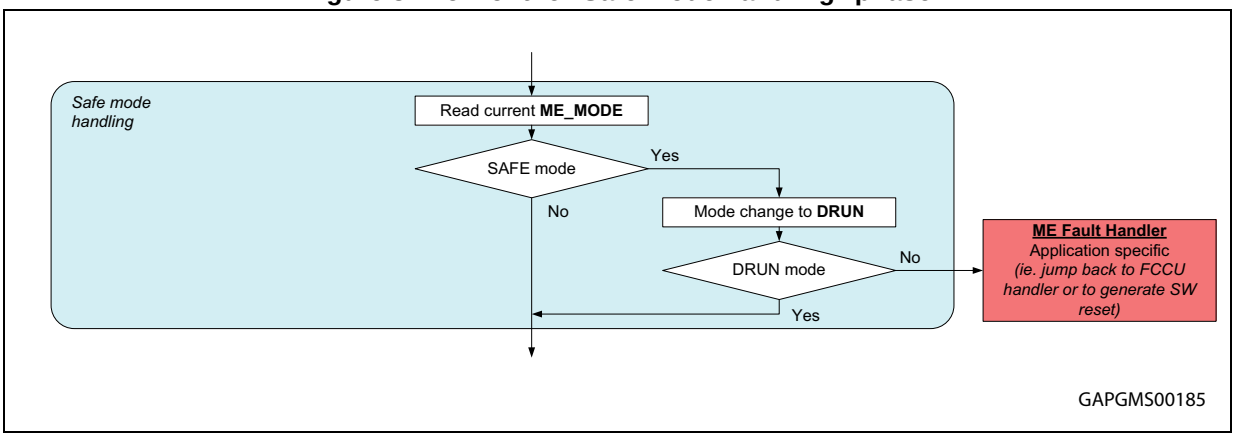
If the functional reset was unexpected, the next steps depends on the application. Unexpected reset always means a sudden critical problem that had happened in the microcontroller.

Some of the possibilities of next steps are:

- Continue as it would be destructive reset with complete initialization
- Generate a destructive reset to run STCU built in self tests again to check if there is no new fault present in the microcontroller

6 Safe mode block

Figure 8. Flow of the "Safe mode handling" phase



Safe mode is a special mode of the microcontroller where only limited part of the hardware functionality is enabled to the user. All attempts to change the HW configuration are blocked until fault conditions are removed.

Very often safe mode is observed in the startup phase after a functional reset other then functional reset initiated from software. In such situation startup has to leave safe mode prior to any mode change transition.

Conditions to leave safe mode are:

- No FCCU fault pending
- FCCU flags in RGM.FES register cleared

6.1 Read current mode of the microcontroller

Current mode of the microcontroller is reported in Mode Entry Global Status (ME.GS) register.

6.1.1 Code sequence

```
1.  32 bit read of ME.GS register (0xC3FDC000)
```

6.2 Safe mode check

It is needed to extract S_CURRENT_MODE bit field from the read value. There are only two possible values after the reset:

- 2 = SAFE
- 3 = DRUN

6.2.1 Code sequence

1. Extract the S_CURRENT_MODE bit field
2. Check if the value is equal to 2 (safe mode)
3. If Yes > go to mode transition step, otherwise continue with next startup block

6.3 Mode change to DRUN request

Mode change request consist of a sequence of two 32 bit writes instruction containing requested mode (3=DRUN) together with a key, where the second key is the inverted value of the first one.

It is necessary to wait for the end of the transition. Ongoing transition is reported by MTRANS bit in ME.GS register.

6.3.1 Code sequence

1. Write value 0x30005AF0 (DRUN + 1.key) to ME.MCTL register (0xC3FDC004)
2. Write value 0x3000A50F (DRUN + 2.key) to ME.MCTL register (0xC3FDC004)
3. Read ME.GS register
4. Check MTRANS bit, if '1' > go back to step 3 (Read ME.GS register)

6.4 DRUN mode check

If there is no pending FCCU fault and FCCU reset flags in the RGM module are cleared, mode transition should finish successfully. If not, a fault handler must handle this problem.

S_CURRENT_MODE bit field should equal to number 3 (DRUN).

6.4.1 Code sequence

1. Read ME.GS register
2. Extract S_CURRENT_MODE bit field
3. Check if the value equals to 3 (DRUN), if No > go to ME_Fault_handler

6.5 ME fault handler

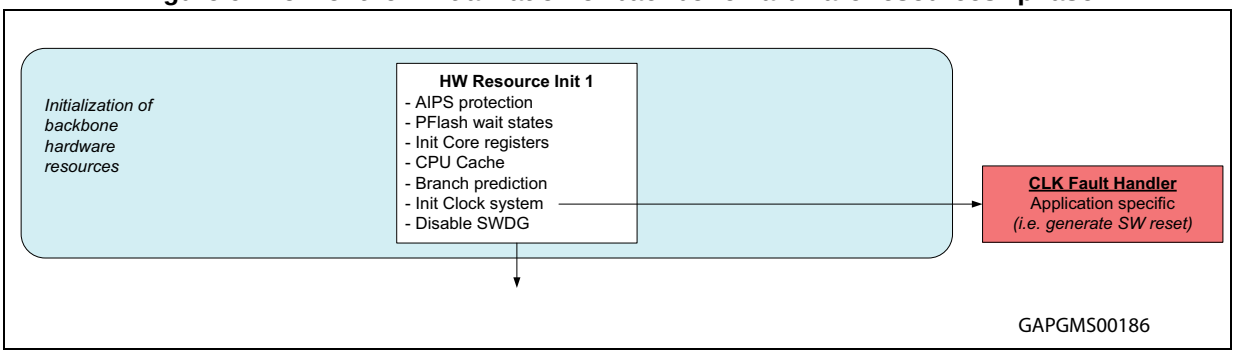
ME_Fault_handler shall handle the situation when a requested transition from safe mode to DRUN mode fails and the microcontroller remains in the safe mode. A reason can be an unsuccessful clear operation of FCCU faults or the presence of another hardware fault.

In such case the decision for the next steps is user specific. Some possibilities can be:

- Go back to FCCU step to re-run again through FCCU and RGM blocks.
- Force the reset to start the device from the very beginning of HW reset initialization, which might be appropriate if the transition failure was because of undefined temporary fault.

7 HW resource initialization

Figure 9. Flow of the "Initialization of backbone hardware resources" phase



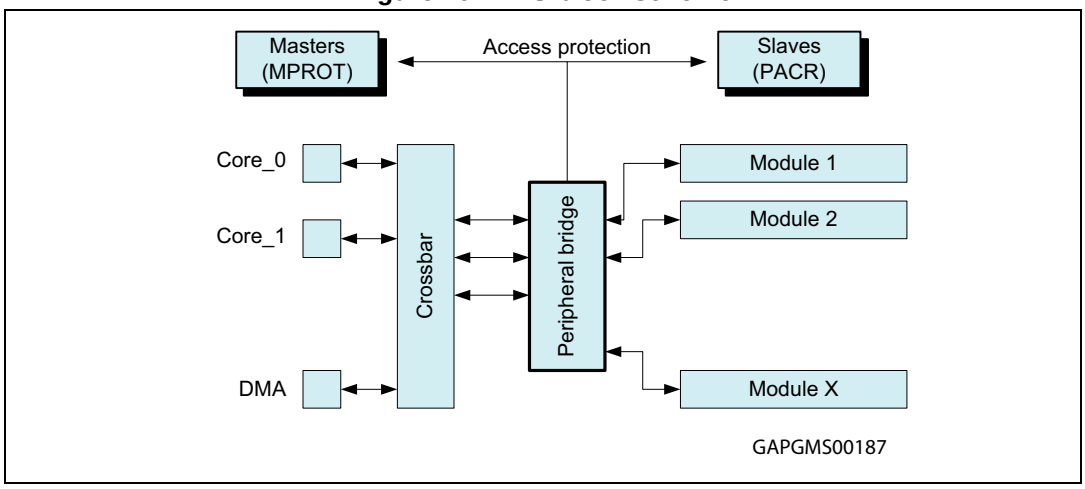
This block is responsible for the initialization of the hardware resources that have an impact on startup time, like optimizing the flash access time, initialization of the core cache, setting up clock subsystem etc.

Anything that does not have an impact on the startup should be left on the application startup executed later on.

7.1 AIPS protection

AIPS protection is a feature implemented in a peripheral bridge module to control access rights of bus masters accessing peripheral modules.

Figure 10. AIPS block scheme



7.1.1 Default reset configuration

- Core_0/1 Both are privileged and trusted to access peripherals.
- DMA,Flexray Not privileged, forced to an user mode. They cannot access peripheral memory space without modification of AIPS configuration.

If the startup relies on a code execution from the core, there is no need to change the configuration of the AIPS. But when the startup wants to use DMA capability, it is necessary to change *MPROT.MPL* bit field.

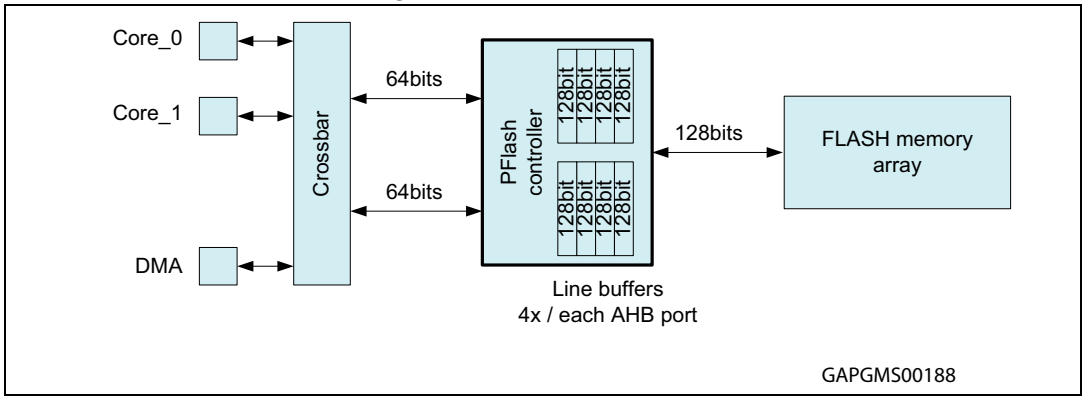
Code sequence

```
1.  Write value 0x77770070 to AIPS_MPROT register
```

7.2 PFlash controller setup

PFlash controller acts as an interface between masters (AHB bus) and a Flash memory array. It controls timing access time and buffering scheme for instruction fetches and data read.

Figure 11. PFlash controller



7.2.1 Default reset configuration

- Instruction fetch — Buffers enabled (3x for instruction fetch), prefetch enabled
- Data read — Buffer disabled
- Access timing — 3 wait states

Default setting is optimal for startup code execution that will run on 120 MHz clock. An assumption is that the startup code does not use many constants placed in Flash as data buffer is disabled.

Note: PFlash register modification shall be done when executing the code from RAM

7.2.2 Code sequence

No need to change default reset configuration.

7.3 Core registers initialization

When running the Leopard device in Lock-Step mode (LSM) it is important to initialize some of the Core registers to a value that are identical for both Cores. The reason behind this is that some Core registers are un-initialized by the reset and it can happen that under certain

circumstances different values in both cores can lead to the reset because of RCCU (checker blocks) event.

In decoupled mode, it is not necessary to have this explicit initialization.

7.3.1 Code sequence

No need to initialize core registers.

7.4 Cache memory initialization

Cache memory initialization does not bring any advantage for a simple startup code with a sequential flow and small loops that fit into buffer lines of PFlash controller. Default PFlash setup can store up to twelve 32 bit instruction or twenty-four 16 bit instructions.

Of course cache memory should be enabled somewhere in the startup code, because it brings significant increase in performance (~30 %). If the startup code is the place of initialization, cache enable bit in L1CSR1 special function register must be set.

7.4.1 Default reset configuration

- Cache disabled

7.4.2 Code sequence

```
1. Enable cache (bit ICE) in Core L1CSR1 special function register
```

7.5 Branch prediction

Z4 core implemented on the SPC564Lx device contains simple branch target buffer prediction feature that brings small performance increase for an application code.

It is not really needed to be done in the startup code, but if the startup is the place of initialization, code shall enable the branch prediction in *BUCSR* special purpose register.

7.5.1 Default reset configuration

- Branch target buffer prediction disabled

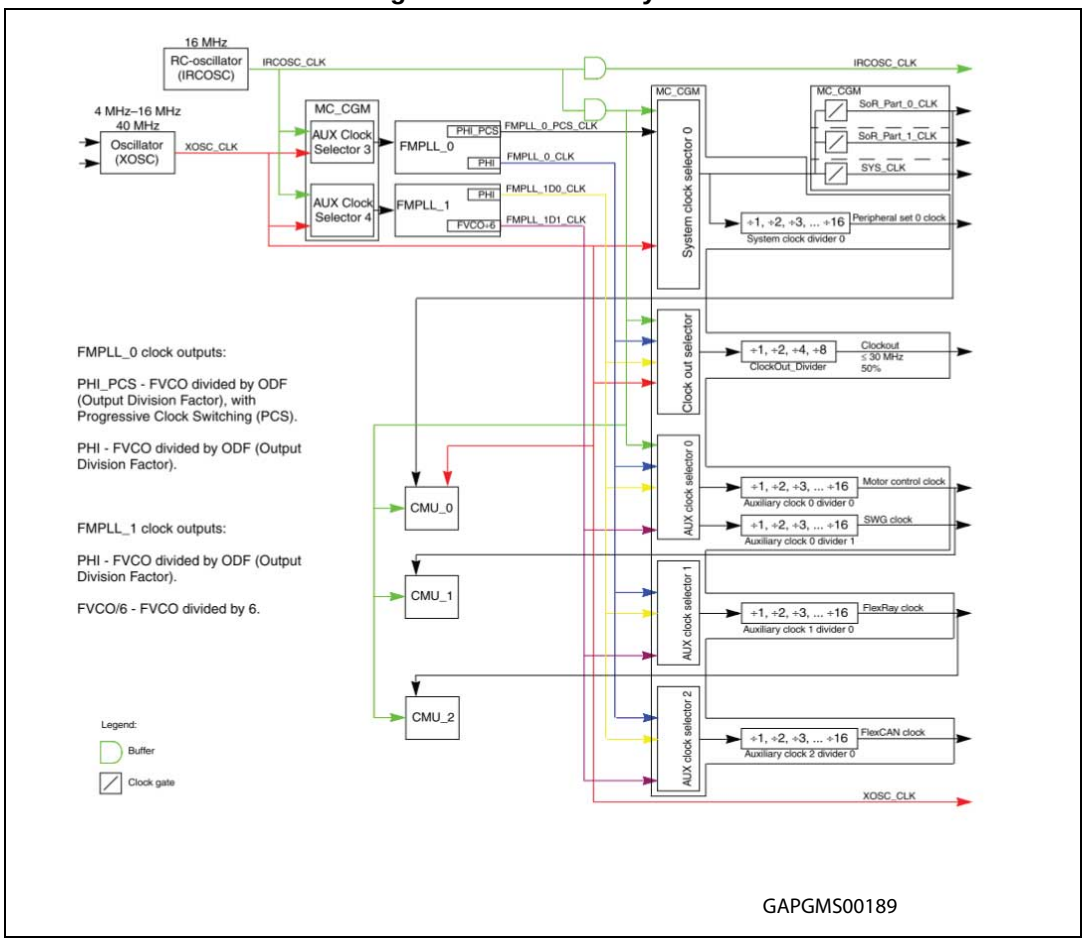
7.5.2 Code sequence

```
1. Set BPEN bit in BUCSR core register
```

7.6 Clock system initialization

Clock subsystem consists of clock generators, clock selectors with dividers and clock monitor units.

Figure 12. Clock subsystem



7.6.1 Default reset configuration

- System clock — IRC_16 MHz
- Oscillator — Disabled
- FMPLLs — Disabled
- Clock output — Disabled
- Peripheral clock — Enabled, PCLK = IRC_16 MHz
- Motor clock — Enabled, MCLK = IRC_16 MHz
- Flexray clock — Enabled, FCLK = system FMPLL
- CAN clock — Enabled, CCLK = system FMPLL
- FMPLL0 input — IRC_16 MHz
- FMPLL1 input — IRC_16 MHz

As seen from the default configuration above, the system clock used for code execution is the internal IRC oscillator running on 16 MHz. It is advised to change the clock setting before RAM initialization and before the application startup to make the process faster.

It is recommended to split clock subsystem setup to several parts to make the clock transition safe with minimum impact on voltage regulator load. All mode transitions issued here use DRUN mode as a target mode.

1. Configure oscillator module
2. Enable oscillator block (mode transition)
3. Configure input clock to FMPLL0
4. Configure FMPLL0
5. Enable FMPLL0 block (mode transition)
6. Wait for FMPLL0 lock
7. Enable peripherals needed in the startup (clock gating)
8. Switch system clock to FMPLL0 output (mode transition)

Configure oscillator block

Here one parameter shall be tuned with respect to ECU design and selected crystal oscillator. It is end of count value parameter, *EOCV* bit field in *RC_CTL* register.

- Count value to reach = EOCV*512 — This parameter tells how many clock pulses will be counted before an external crystal clock can be considered as stable and possible to select as the system clock.

EOCV value shall satisfy the following formula:

EOCV >= (TXOSCHSSU * fXOSCHS) / 512

where fXOSCHS is the "oscillator frequency" and TXOSCHSSU is the "oscillator start-up time" whose value can be found in the DS.

Default reset value

- EOCV = 128 — It gives 1.63 ms delay for 40 MHz crystal oscillator

Code sequence

1. `Write required value to EOCV bit field in OSC_CTL register`

Enable oscillator block (mode transition)

In order to enable the crystal oscillator block, mode transition is required with bit XOSCON in ME_DRUN_MC register set on.

Code sequence

1. `Set bit XOSCON in ME_DRUN_MC register`
2. `Write DRUN with Key_1 in ME_MCTL register`
3. `Write DRUN with Key_2 in ME_MCTL register`
4. `Wait mode transition to finish (ME_GS.MTRANS bit = 0)`

Configure input clock to FMPLL0

External crystal oscillator shall be used as input to FMPLL block. A selection is done in the *AUX_Clock selector 3* block.

Default reset value

- FMPLL0 input IRC_16 MHz. It is necessary to change it to XOSC.

Code sequence

1. `Set SELCTL bit field of CGM_AC3_SC register to 1 (XOSC)`

Configure FMPLL0

The FMPLL0 configuration consists in setting right values to, IDF, ODF and NDIV, in FMPLL0.CR register. Values have to fulfill equations given in RM0032 (see *B.1: Reference documents*). Their values depend on selected crystal oscillator and target system frequency.

Default reset values

- IDF=1 Input frequency divided by 2. It must be changed to fulfill 4-16 MHz input range in case of 40 MHz crystal oscillator
- ODF=1 Divide VCO output frequency by 4.
- NDIV=64 Input frequency divided by IDF block multiplied by NDIV to get VCO frequency, which must be within 256-512 MHz frequency range.

Example 1

Setting for 120 MHz system clock derived from 40 MHz crystal oscillator

- IDF=3 40 MHz / 4 = 10 MHz as input to FMPLL
- NDIV=48 10 MHz * 48 = 480 MHz as VCO frequency
- ODF=1 480 MHz / 4 = 120 MHz system clock

Code sequence

1. `Write IDF,ODF and NDIV values to FMPLL0.CR register (0xC3FE_00A0)`

Enable FMPLL0 block (mode transition)

It is necessary to set *PLL0ON* bit in *ME_DRUN_MC* register and then to initiate a mode transition to DRUN.

Code sequence

1. `Set bit PLL0ON in ME_DRUN_MC register`
2. `Write DRUN with Key_1 in ME_MCTL register`
3. `Write DRUN with Key_2 in ME_MCTL register`
4. `Wait mode transition to finish (ME_GS.MTRANS bit = 0)`

Wait for FMPLL0 lock

Before FMPLL0 can be used as a system clock it is necessary to wait for the PLL lock event. CR register in the FMPLL0 block contains S_LOCK bit that tells the LOCK status:

- 0 = FMPLL unlocked
- 1 = FMPLL locked

Usual lock time delay is around 200 µs.

Code sequence

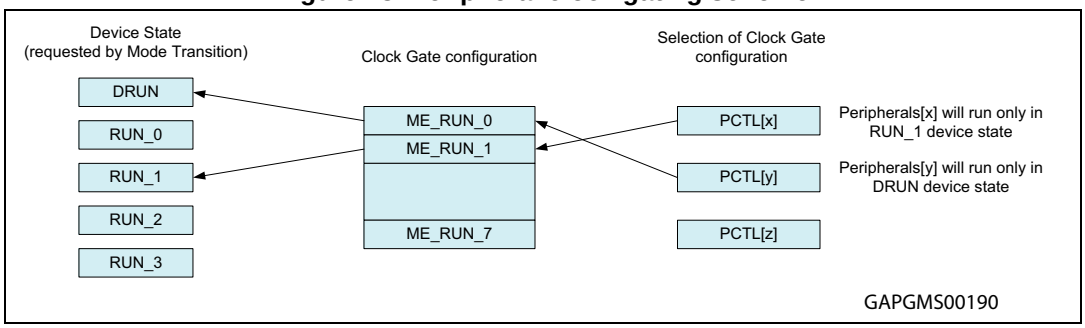
1. `Read FMPLL0 CR register`
2. `Test s_lock bit`
3. `If '0' > wait loop to step 1`

Enable peripherals needed in the startup

There is a set of peripherals that are clock gated (all peripherals having PCTL configuration register inside mode entry module). Each peripheral within this group has to be explicitly configured to run in dedicated mode. This configuration is done via pair of registers, *ME_RUN_PC0-7* and *ME_PCTLx*.

ME_RUN_PC[0-7] registers indicate if the peripherals are active and when they are inactive (clock gated). The peripheral clock for a given mode transition must be configured by the user.

Figure 13. Peripheral clock gating scheme



Default reset state

- ME_RUN_PCx — No mode selected, no clock enabled to any gated peripheral
- PCTLx — They point to ME_RUN_PC0

Note: Any change in ME_RUN_PC0 will be used by all peripherals if they are not reconfigured to use another ME_RUN_PCx register. Result can be that all peripherals, even peripherals that are not needed, are enabled if ME_RUN_PC0 is enabled for current device state mode.

Good practice is not to use ME_RUN_PC0 register, but to configure one of the ME_RUN_PC[1-7] and configure needed peripherals accordingly.

Code sequence

1. Set DRUN bit in ME_RUN_PC1 register (enable DRUN in ME_RUN_PC1 configuration)
2. Set RUN_CFG bit field to 1 in ME_PCTL register of peripherals to be activated
3. Peripherals will be activated with next Mode Transition to DRUN

Switch system clock to FMPLL0 (mode transition)

In order to change system clock another mode transition is required. This transition will be used for enabling the clock gated peripherals that were prepared in previous step.

Code sequence

1. Set SYSCLK field to 6 in ME_DRUN_MC register (PLL as system clock)
2. Write DRUN with Key_1 in ME_MCTL register
3. Write DRUN with Key_2 in ME_MCTL register
4. Wait mode transition to finish (ME_GS.MTRANS bit = 0)
5. Read ME.GS register and check that SYSCLK field equals to 4 (FMPLL0)
6. If not > go to CLK_Fault_Handler

7.7 CLK fault handler

It is up to an application what to do in case of a problem with system clock initialization. There might be several reasons for the failure of the initialization

- Crystal oscillator does not start
- Crystal oscillator starts at wrong frequency
- FMPLL0 does not lock (here software watchdog expires and trigger reset earlier than going to CLK fault handler)

The standard solution might be to ask for reset to try initialization from the very beginning again.

7.8 Software Watchdog Timer (SWT) disable

Depending on the Option byte configuration in the shadow Flash memory the software watchdog is enabled after the reset or disabled. SWT module always runs on IRC_16 MHz clock base.

It is recommended to leave it enabled after the reset up to this point as it monitors progression of the startup code and hardware initialization. Any fault leading to lockup situation in the previous steps will finish in a reset event caused by SWT expiration event.

The reason why the software watchdog is going to be disabled here is time needed for HW initialization (mainly contribution of clock initialization) plus RAM memory initialization that takes significant time. Both contributors can take longer than it is a default SWT expiration time.

7.8.1 Default reset value

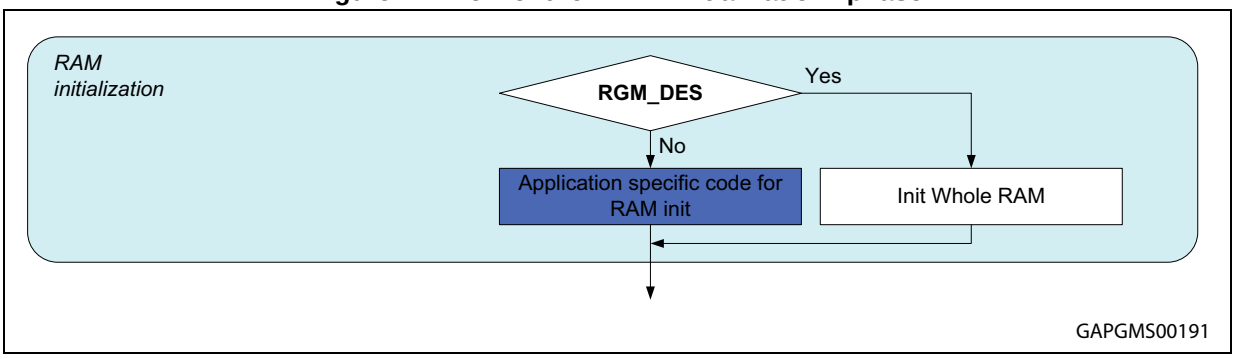
- SWT expiration ~16 ms

7.8.2 Code sequence

1. Clear *SLK* (soft lock bit) in *CR* register by sequence of two writes to *SR* register using proper keys
2. Clear *WEN* bit in *CR* register

8 RAM initialization

Figure 14. Flow of the "RAM initialization" phase



An Important part of the startup code is linked to RAM initialization. Especially in case of destructive resets when the content of the RAM cells, data plus ECC syndrome bits, are undefined. In such case the entire RAM memory or at least all used sections must be initialized to update ECC syndrome bits. Not doing this can lead to multiple bit ECC faults during any read operation or byte, half word write operation.

Default reset state

- Destructive reset RAM undefined. It is needed to initialize it completely.
- Functional reset RAM preserves its content. It can be used as non-volatile storage.

8.1 Init whole RAM

There are mainly two ways how to initialize the RAM. Both need supportive information from linker file.

- Using multiple word store instructions `e_stmw`, that store set of GPRs registers in consecutive order
- Using standard 32 bit store instructions `e_stwu`

This part of startup code can be used for initializing the RAM memory to specific value, i.e. zero.

8.1.1 Code sequence

1. Determine number of cycles to execute in multiples of words. It depends on type of instructions.
2. Place iteration count to core counter register
3. Prepare start address and adjust it if *e_stwu* instruction will be used
4. Prepare value to write. In case of *e_stmw* instruction, the value must be written to all involved GPRs
5. Issue store instruction
6. Decrement counter and jump for next store cycle if not zero

9 Application ‘asm’ startup

Before jumping to ‘C’ language to continue with the startup there are still some steps required. These steps are completely under user or any other software vendor (i.e. OS supplier) responsibility. Simply this ‘asm’ part shall prepare a base for the ‘C’ programming environment.

Among usual steps belongs:

- Initialization of r1 register (link register)
- Initialization of r2 register (sdata2 base pointer)
- Initialization of r13 register (sdata base pointer)
- Stack initialization
- Initialization of initialized variables

10 Application ‘C’ startup

High level language application startup is under user responsibility.

Examples of application ‘C’ startup routines are:

- External hardware initialization
- EEPROM control block initialization, reading stored data
- Software layers initialization
- Application blocks initialization

Part of the application startup shall be safety checks. Safety checks related to microcontroller hardware are described more in AN3324 (see *B.1: Reference documents*).

Appendix A asm examples

All ‘asm’ examples here are built with WindRiver compiler toll with preprocessing on assembly files set on. Preprocessor directives are used to make the code more readable. Readable code is prioritized to time or space optimization.

A.1 Symbol defines

```
// MSR value to set after the reset
#define STARTUP_MSR 0x21200 // CE|ME|DE

// Software watchdog module
#define SWT_BASE        0xFFF38000 // Software watchdog base address
#define SWT_REG_CR      0x0000     // registers offset
#define SWT_REG_SR      0x0010
#define SWT_UNLOCK_KEY1 0xC520     // key1 for soft_lock unlock
#define SWT_UNLOCK_KEY2 0xD928     // key2 for soft_lock unlock

// FLASH registers
#define FLASH_BASE     0xC3F88000 // Flash registers base address
#define FLASH_REG_MCR  0x0000     // registers offset
#define FLASH_EER_MASK 0x00008000 // EER bit

// FCCU module constants
#define FCCU_BASE         0xFFE6C000 // FCCU module base address
#define FCCU_REG_CTRL     0x0000     // registers offset
#define FCCU_REG_CF       0x006C
#define FCCU_REG_CFK      0x007C
#define FCCU_REG_NCF      0x0080
#define FCCU_REG_NCFK     0x0090
#define FCCU_REG_STATUS   0x00C0
#define FCCU_REG_IRQ      0x00E0
#define FCCU_OPR_STATUS   0x3        // command to read Status
#define FCCU_OPR_CF_READ  0x9        // command to read CF
#define FCCU_OPR_NCF_READ 0xA        // command to read NCF
#define FCCU_CFK_KEY      0x618B7A50 // key for CF faults clear operation
#define FCCU_NCFK_KEY     0xAB3498FE // key for NCF faults clear operation
#define FCCU_OPS_SUCCESS  3          // command successful status to wait for
#define FCCU_CF_STCU      0x0850     // upper half word for compare instruction
#define FCCU_NCF12        19         // BSTI instr has inverse bit numbering

// RGM module constants
#define RGM_BASE          0xC3FE4000 // Reset module base address
#define RGM_REG_FESDES    0x0000     // register offset from RGM_BASE
#define RGM_DES_MASK      0x8000FFFF // Destructive resets status bit mask

// MODE ENTRY module constants
#define ME_BASE         0xC3FDC000 // Mode Entry base address
#define ME_REG_GS       0x0000     // register offsets from ME_BASE
#define ME_REG_MCTL     0x0004
#define ME_REG_DRUN_MC  0x002C
#define ME_REG_RUN_PC1  0x0084
#define ME_REG_PCTL_CRC 0x00FA
#define ME_GS_MODE_SAFE 2          // current mode SAFE
```

```
#define ME_GS_MODE_DRUN   3          // current mode DRUN
#define ME_GS_MTRANS      4          // position of bit for e_btsti instr.
#define ME_GS_SYSCLK_MASK 0x000F     // mask of system clk field
#define ME_MCTL_DRUN_R1   0x30005AF0 // 1.step in mode transition request
#define ME_MCTL_DRUN_R2   0x3000A50F // 2.step in mode transition request
#define ME_MC_XOSCON      26         // bit position for se_bseti instr.
#define ME_MC_PLL0ON      25         // bit position for se_bseti instr.
#define ME_MC_SYSCLK_PLL  0x0004     // value for e_or2i instr.
#define ME_RUNPC_DRUN     28         // bit position for se_bseti instr.
#define ME_PCTL_RUNPC1    1          // pointer to RUN_PC1 configuration

// AIPS peripheral bridge
#define AIPS_BASE         0xFFF00000 // peripheral bridge base address
#define AIPS_REG_MPROT    0x0000     // register offset from AIPS_BASE
#define AIPS_MPROT_ENABLE 0x77770070 // Core0,1, DMA0, FlexRay as superuser

// Core Cache memory
#define CACHE_REG_L1CSR1   1011   // SPR register number
#define CACHE_ICE_BIT      0x0001 // for e_li instruction

// CGM block
#define CGM_BASE       0xC3FE0000 // base address of CGM module
#define CGM_REG_AC3SC  0x0398     // register offset from CGM_BASE
#define CGM_AC3SC_XOSC 0x0100     // value for PLL input AUX3 bridge

// XOSC block (CGM base)
#define XOSC_REG_OSCCTL 0x0000   // register offset from CGM_BASE
#define XOSC_EOCV_IMASK 0xFF00   // mask to clear EOCV default value
#define XOSC_EOCV_VALUE 0x009C   // cca 2ms for 40MHz crystal

// FMPLL0 block (CGM base)
#define FMPLL0_REG_CR 0x00A0     // register offset from CGM_BASE
#define FMPLL0_SLOCK  28         // bit position for e_btsti instr.

#define FMPLL0_IDF    0x0C00     // 3 = divide by 4
#define FMPLL0_ODF    0x0100     // 1 = divide by 4
#define FMPLL0_NDIV   0x0030     // 48 = multiply by 48
#define FMPLL0_VALUE  (FMPLL0_IDF | FMPLL0_ODF | FMPLL0_NDIV)
```

A.2 Core register init (LSM)

```
// selected core SPRs
mtcrf 0xFF,r31 // Condition register
mtspr 285,r31 // TBU
mtspr 284,r31 // TBL
mtspr 272,r31 // SPRG0
mtspr 273,r31 // SPRG1
mtspr 274,r31 // SPRG2
mtspr 275,r31 // SPRG3
mtspr 276,r31 // SPRG4
mtspr 277,r31 // SPRG5
mtspr 278,r31 // SPRG6
mtspr 279,r31 // SPRG7
mtspr 604,r31 // SPRG8
mtspr 605,r31 // SPRG9
```

```
mtspr 26,r31 // SRR0
mtspr 27,r31 // SRR1
mtspr 58,r31 // CSRR0
mtspr 59,r31 // CSRR1
mtspr 63,r31 // IVPR
mtspr 61,r31 // DEAR
mtspr 62,r31 // ESR
mtspr 570,r31 // MCSRR0
mtspr 571,r31 // MCSRR1
mtspr 340,r31 // TCR
mtspr 512,r31 // SPEFSCR
mtspr 1,r31   // XER
mtspr 256,r31 // USPRG0
mtspr 9,r31   // CTR
mtspr 8,r31   // LR
mtspr 308,r31 // DBCR0
mtspr 309,r31 // DBCR1
mtspr 310,r31 // DBCR2
mtspr 561,r31 // DBCR3
mtspr 563,r31 // DBCR4
mtspr 564,r31
mtspr 603,r31
```

A.3 MMU entry configuration

```
// Table 0
e_lis   r0, 0x1000
e_or2i  r0, 0x0000
mtspr   mas0, r0     // fix bits + TLB entry
e_lis   r0, 0xC000
e_or2i  r0, 0x0500
mtspr   mas1, r0     // cfg bits + Page SIZE
e_lis   r0, 0x0000
e_or2i  r0, 0x0028
mtspr   mas2, r0     // EPN + page atrib
e_lis   r0, 0x0000
e_or2i  r0, 0x003F
mtspr   mas3, r0     // RPN + access atrib
tlbwe
```

A.4 SSCM HW configuration check

```
// Base address of FLASH module to r7
e_lis   r7, FLASH_BASE@h
e_or2i  r7, FLASH_BASE@l

// Read MCR register
e_lwz   r0, FLASH_REG_MCR (r7) // read FLASH.MCR register

// MCR.EER bit?
e_lis   r1, FLASH_EER_MASK@h
e_or2i  r1, FLASH_EER_MASK@l
se_and  r0, r1                 // look for destructive resets
se_cmpi r0,0                   // is value '1' ?
```

```
se_bc   00,02,sscm_error_handler     // if yes - jump to SSCM Error handler
...

sscm_error:
// user code - handler of SSCM error
se_b .                               // break due to SSCM fault
```

A.5 FCCU handling block

```
// Base address of FCCU module to r7
e_lis    r7, FCCU_BASE@h
e_or2i   r7, FCCU_BASE@l

// Read Status
e_li     r0, FCCU_OPR_STATUS    // prepare command
e_stw    r0, FCCU_REG_CTRL (r7) // issue command
.read_status:
e_lwz    r0, FCCU_REG_CTRL (r7) // read FCCU_CTRL register
e_rlwinm r0,r0,26,30,31         // rotate OPS field and mask it
se_cmpi  r0, FCCU_OPS_SUCCESS   // look for OPS = SUCCESS
se_bc    00,02,.read_status     // wait to finish the command
e_lwz    r5, FCCU_REG_STATUS (r7) // read Status
se_cmpi  r5,0                     // Status == NORMAL?
se_bc    1,2,fccu_no_fault        // no fault in fccu > skip fault handling

// Read CF faults
e_li     r0, FCCU_OPR_CF_READ     // prepare command
e_stw    r0, FCCU_REG_CTRL (r7)   // issue command
.read_cf:
e_lwz    r0, FCCU_REG_CTRL (r7)   // read FCCU_CTRL register
e_rlwinm r0,r0,26,30,31           // rotate OPS field and mask it
se_cmpi  r0, FCCU_OPS_SUCCESS     // look for OPS = SUCCESS
se_bc    00,02,.read_cf           // wait to finish the command
e_lwz    r3, FCCU_REG_CF (r7)     // read CF register to r3 register
mtsprg0  r3                       // save them to Core SPRG0 register

// Read NCF faults
e_li     r0, FCCU_OPR_NCF_READ    // prepare command
e_stw    r0, FCCU_REG_CTRL (r7)   // issue command
.read_ncf:
e_lwz    r0, FCCU_REG_CTRL (r7) // read FCCU_CTRL register
e_rlwinm r0,r0,26,30,31         // rotate OPS field and mask it
se_cmpi  r0, FCCU_OPS_SUCCESS   // look for OPS = SUCCESS
se_bc    00,02,.read_ncf        // wait to finish the command
e_lwz    r4, FCCU_REG_NCF (r7)  // read NCF register to r4 register
mtsprg1  r4                     // save them to Core SPRG1 register

// Check STCU faults
se_mr     r0,r3                 // take copy of CF faults
e_and2is. r0,0x0850             // mask bits CF[27,22,20]
se_cmpi   r0,0                  // is one of them '1' ?
se_bc     00,02,fccu_stcu_error // if yes - jump to error handler
se_btsti  r4, FCCU_NCF12        // check bit NCF[12]
se_bc     1,2,fccu_analyze      // jump if all STCU bits are '0'
```

```
fccu_stcu_error:
// user code - handler of STCU errors
se_b .                          // break due to STCU faults

fccu_analyze:
// user code - analyze FCCU faults, if there are any
// ...
se_b      fccu_clear            // continue with clear operation

fccu_clear:
// clear FCCU Critical faults CF
e_lis   r0, FCCU_CFK_KEY@h
e_or2i  r0, FCCU_CFK_KEY@l
e_stw   r0, FCCU_REG_CFK (r7)   // write Key to clear CF faults
e_stw   r3, FCCU_REG_CF (r7)    // clear known CF faults
.wait_cf_clear:
e_lwz    r0, FCCU_REG_CTRL (r7) // read FCCU_CTRL register
e_rlwinm r0,r0,26,30,31         // rotate OPS field and mask it
se_cmpi  r0, FCCU_OPS_SUCCESS   // look for OPS = SUCCESS
se_bc    00,02,.wait_cf_clear   // wait to finish the command

// clear FCCU Non-Critical faults (NCF)
e_lis    r0, FCCU_NCFK_KEY@h
e_or2i   r0, FCCU_NCFK_KEY@l
e_stw    r0, FCCU_REG_NCFK (r7) // write Key to clear CF faults
e_stw    r4, FCCU_REG_NCF (r7)  // clear known NCF faults
.wait_ncf_clear:
e_lwz    r0, FCCU_REG_CTRL (r7) // read FCCU_CTRL register
e_rlwinm r0,r0,26,30,31         // rotate OPS field and mask it
se_cmpi  r0, FCCU_OPS_SUCCESS   // look for OPS = SUCCESS
se_bc    00,02,.wait_ncf_clear  // wait to finish the command

// continue with next block
se_b     fccu_end

// no FCCU faults
fccu_no_fault:
e_lis r0,0
mtsprg0 r0                     // save no CF fault to Core register
mtsprg1 r0                     // save no CF fault to Core register
fccu_end:
```

A.6 RGM block

```
// Base address of RGM module to r7
e_lis  r7, RGM_BASE@h
e_or2i r7, RGM_BASE@l

// Read all RGM flags (FES+DES)
e_lwz   r0, RGM_REG_FESDES (r7)
mtsprg2 r0                              // save reset flags to Core SPRG2 register

// destructive reset ?
e_lis   r1, RGM_DES_MASK@h
e_or2i  r1, RGM_DES_MASK@l
```

```
se_and  r0, r1                          // look for destructive resets
se_cmpi r0,0                            // is one of them '1' ?
se_bc   01,02,rgm_fes_handler           // if no - jump to Functional handler
rgm_des_handler:
// do what is needed for destructive path
// ....
se_b    rgm_clear                       // go to flag reset operation

rgm_fes_handler:
// analyze if the reset was expected or not
// ...
rgm_clear:
e_lis   r1, 0xFFFF
e_or2i  r1, 0xFFFF                    // prepare value for clear-write-1 oper.
e_stw   r1, RGM_REG_FESDES (r7)       // write '1' to clear all flags
rgm_end:
```

A.7 Safe mode block

```
// Base address of ME module to r7
e_lis     r7, ME_BASE@h
e_or2i    r7, ME_BASE@l

// read status & check SAFE mode
e_lwz     r0, ME_REG_GS (r7)            // read GS register
se_srwi   r0, 0x1C                      // shift S_CURRENT_MODE field for comparison
se_cmpi   r0, ME_GS_MODE_SAFE           // compare with SAFE mode value
se_bc     0,2,me_end                    // not in SAFE mode > end SAVE handler

me_save_drun_change:
// SAFE mode is present, request change to DRUN
e_lis     r7, ME_BASE@h
e_or2i    r7, ME_BASE@l
e_lis     r0, ME_MCTL_DRUN_R1@h
e_or2i    r0, ME_MCTL_DRUN_R1@l
e_stw     r0, ME_REG_MCTL (r7)          // 1. step - normal key
e_lis     r0, ME_MCTL_DRUN_R2@h
e_or2i    r0, ME_MCTL_DRUN_R2@l
e_stw     r0, ME_REG_MCTL (r7)          // 2. step - inverted key

// wait for transition to complete (test of GS.MTRANS bit)
me_wait_transition:
e_lwz     r0, ME_REG_GS (r7)            // read GS register
se_btsti  r0,ME_GS_MTRANS               // test the MTRANS bit
se_bc     0,2,me_wait_transition        // if '1' > wait in loop

// read status & check DRUN mode
se_srwi   r0, 0x1C                      // shift S_CURRENT_MODE rigth for comparison
se_cmpi   r0, ME_GS_MODE_DRUN           // compare with DRUN value
se_bc     1,2,me_end                    // if DRUN=TRUE > end the ME handler

// DRUN is not entered
// execute Fault management
me_drun_fault:
// execute fault handler in the case of unsuccessful Mode transition
```

```
se_b .            // here infinitive loop

me_end:
```

A.8 AIPS protection

```
aips_start:
e_lis     r7, AIPS_BASE@h
e_or2i    r7, AIPS_BASE@l           // r7 = AIPS bridge base address
e_lis     r0, AIPS_MPROT_ENABLE@h
e_or2i    r0, AIPS_MPROT_ENABLE@l   // value to write to AIPS MPROT register

se_stw    r0, AIPS_REG_MPROT (r7)      // write the value
```

A.9 Cache memory initialization

```
init_cache:
e_li    r0, CACHE_ICE_BIT    // prepare ICE bit to enable the cache
mtspr   CACHE_REG_L1CSR1, r0 // enable Cache
```

A.10 Branch prediction

```
mfbucsr  r3       // read BUCSR special function register
e_or2i   r3, 0x1  // set branch prediction bit
mtbucsr  r3       // write value back to enable it
```

A.11 Clock system initialization

```
// setup CGM and ME base addresses
e_lis    r6, CGM_BASE@h
e_or2i   r6, CGM_BASE@l
e_lis    r7, ME_BASE@h
e_or2i   r7, ME_BASE@l

// setup EOCV value in XOSC block
e_lhz    r0, XOSC_REG_OSCCTL (r6) // get half word with EOCV bit field
e_and2i. r0, XOSC_EOCV_IMASK      // clear EOCV field
e_or2i   r0, XOSC_EOCV_VALUE      // set new value
e_sth    r0, XOSC_REG_OSCCTL (r6) // write it back

// enable XOSC in DRUN mode
e_lwz    r0, ME_REG_DRUN_MC (r7)  // read ME_DRUN_MC register
se_bseti r0, ME_MC_XOSCON         // set on XOSCON bit
e_stw    r0, ME_REG_DRUN_MC (r7)  // write value back

// mode transition -> DRUN
e_lis    r0, ME_MCTL_DRUN_R1@h
e_or2i   r0, ME_MCTL_DRUN_R1@l
e_stw    r0, ME_REG_MCTL (r7)     // 1. step - normal key
e_lis    r0, ME_MCTL_DRUN_R2@h
e_or2i   r0, ME_MCTL_DRUN_R2@l
```

```
e_stw    r0, ME_REG_MCTL (r7)     // 2. step - inverted key

// wait for transition to complete (test of GS.MTRANS bit)
me_wait_xoscon:
e_lwz    r0, ME_REG_GS (r7)       // read GS register
se_btsti r0, ME_GS_MTRANS         // test the MTRANS bit
se_bc    0,2,me_wait_xoscon       // if '1' > wait in loop

// select XOSC as input to FMPLL0 block
e_li     r0, CGM_AC3SC_XOSC       // prepare value
e_sth    r0, CGM_REG_AC3SC (r6)   // write it to CGM_AC3SC register

// configure FMPLL0
e_li     r0, FMPLL0_VALUE         // prepare value for FMPPL0 configuration
e_sth    r0, FMPLL0_REG_CR (r6)   // write it to FMPLL0_CR register
// enable FMPLL0 in DRUN mode
e_lwz    r0, ME_REG_DRUN_MC (r7)  // read actual value of ME_DRUN_MC reg.
se_bseti r0, ME_MC_PLL0ON         // set bit PLL0ON
e_stw    r0, ME_REG_DRUN_MC (r7)  // update ME_DRUN_MC register

// mode transition -> DRUN
e_lis    r0, ME_MCTL_DRUN_R1@h
e_or2i   r0, ME_MCTL_DRUN_R1@l
e_stw    r0, ME_REG_MCTL (r7)     // 1. step - normal key
e_lis    r0, ME_MCTL_DRUN_R2@h
e_or2i   r0, ME_MCTL_DRUN_R2@l
e_stw    r0, ME_REG_MCTL (r7)     // 2. step - inverted key

// wait for transition to complete (test of GS.MTRANS bit)
me_wait_pll0on:
e_lwz      r0, ME_REG_GS (r7)       // read GS register
se_btsti   r0, ME_GS_MTRANS         // test the MTRANS bit
se_bc      0,2,me_wait_pll0on       // if '1' > wait in loop

// wait for PLL lock event
pll_wait_lock:
e_lwz      r0, FMPLL0_REG_CR (r6) // read FMPLL0.CR register
se_btsti   r0, FMPLL0_SLOCK       // test LOCK bit
se_bc      1,2,pll_wait_lock      // if '0' > wait in loop

// enable peripherals needed in next startup steps
e_lwz      r0, ME_REG_RUN_PC1 (r7)  // read value of ME_RUN_PC1 register
se_bseti   r0, ME_RUNPC_DRUN        // set DRUN bit
e_stw      r0, ME_REG_RUN_PC1 (r7)  // write value back
e_li       r0, ME_PCTL_RUNPC1       // prepare value for PCTL
e_stb      r0, ME_REG_PCTL_CRC (r7) // set value to ME_PCTL_CRC register

// select PLL for system clock
e_lwz      r0, ME_REG_DRUN_MC (r7) // read current value of ME_DRUN_MC reg
e_or2i     r0, ME_MC_SYSCLK_PLL    // set PLL as system clock
e_stw      r0, ME_REG_DRUN_MC (r7) // write the value back

// mode transition -> DRUN
e_lis      r0, ME_MCTL_DRUN_R1@h
e_or2i     r0, ME_MCTL_DRUN_R1@l
e_stw      r0, ME_REG_MCTL (r7) // 1. step - normal key
```

```
e_lis      r0, ME_MCTL_DRUN_R2@h
e_or2i     r0, ME_MCTL_DRUN_R2@l
e_stw      r0, ME_REG_MCTL (r7)    // 2. step - inverted key

// wait for transition to complete (test of GS.MTRANS bit)
me_wait_sysclk:
e_lwz    r0, ME_REG_GS (r7)        // read GS register
se_btsti r0, ME_GS_MTRANS          // test the MTRANS bit
se_bc    0,2,me_wait_pll0on        // if '1' > wait in loop

// check that system clock is FMPLL0
e_and2i. r0, ME_GS_SYSCLK_MASK     // get system clock status
se_cmpi  r0, ME_MC_SYSCLK_PLL      // check the value
se_bc    1,2,clk_startup_end       // continue if PLL is the system clock

clk_fault_handler:
se_b .                             // fault handler loop, app code needed

clk_startup_end:
```

A.12 Software Watchdog Timer (SWT) disable

```
e_lis    r7, SWT_BASE@h
e_or2i   r7, SWT_BASE@l         // base address of SWT module
e_li     r0, SWT_UNLOCK_KEY1
e_stw    r0, SWT_REG_SR (r7)    // 1. key
e_li     r0, SWT_UNLOCK_KEY2
e_stw    r0, SWT_REG_SR (r7)    // 2. key -> soft lock disabled
e_lwz    r0, SWT_REG_CR (r7)    // read CR register
se_bclri r0, 31                 // clear WEN bit (disable WEN)
e_stw    r0, SWT_REG_CR (r7)    // write modified value back
```

A.13 Init whole RAM

```
e_lis    r3,__wrs_ramsize_words@h   // size of ECC prot. SRAM in words
e_or2i   r3,__wrs_ramsize_words@l   // = size in bytes /4
mtctr    r3                         // copy this value to counter gpr
e_lis    r7,__wrs_ramstart@h        // r7 is now our ram pointer
e_or2i   r7,__wrs_ramstart@l        // initialized to start of SRAM
se_subi  r7,0x04                    // and now to 1 word location less
e_lis    r3,ECC_RAM_INIT_VALUE@h
e_or2i   r3,ECC_RAM_INIT_VALUE@l
sram_loop:
e_stwu   r3,0x04(r7)                // write r3, to r7+4 and update r7
e_bdnz   sram_loop                  // decr. counter gpr,loop if not 0
```

Appendix B Further information

B.1 Reference documents

- *SPC56EL60 32-bit MCU family built on the embedded Power Architecture*® (RM0032, Doc ID 15265)
- *Using SPC56EL60x fault collection and control unit (FCCU)* (AN4124 Doc ID 023294)
- *Implementing power-on self tests for SPC56EL60 in locked step* (AN3324 Doc ID 18311)

B.2 Acronyms

Table 1. Acronyms

Acronym	Name
DPM	SPC564Lx in Decoupled Parallel mode. Two cores run independently.
LSM	SPC564Lx in Lock Step mode. One core is checked by the other.
RGM	Reset module
FCCU	Fault collection unit
ECSM	Error correction status module
SSCM	System Status and Configuration module.

Revision history

Table 2. Document revision history

Date	Revision	Changes
03-Apr-2013	1	Initial release.
17-Sep-2013	2	Updated Disclaimer.