XC2000 Family

16-bit

Power Management with SCU Driver

AP16170

Application Note

V1.1 2013-08

Microcontrollers

Edition 2013-08

Published by Infineon Technologies AG,

81726 Munich, Germany.

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices, please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Revision History

Major changes since previous revision

Date	Version	Changed By	Change Description
July 2013	1.1		All content revised

We Listen to Your Comments

Is there any information in this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of our documentation. Please send your proposal (including a reference to this document title/number) to:

ctdd@infineon.com

Table of Contents

1 About this document

The microcontrollers of the XC2000/XE166 family come with an enhanced means for system clock control and power management. The SCU Driver, together with the SCU Configuration Tool, provides software that enables the customer to configure and use the features of the System Control Unit (SCU) according to their needs, without detailed knowledge of this powerful and complex unit.

This Application Note describes the SCU driver, the wakeup logic and one application use case (Fast Startup Mode).

The software is based on the Programmer's Guide for Clock and Power Management and on the hardware description.

2 SCU Driver description

The SCU Driver works with all members of the XC2000/XE166 family of products from Infineon

The Driver supports different tool chains, but depending on the controller and tool chain selected, not all features can be used.

Note: For additional information about the SCU Driver, please see also Application Note AP16168.

2.1 Normal Mode

After reset, the system enters Normal (Operation) Mode via the SCU Driver function **Scu_GoFromBaseModeToNormalMode**.

By configuration, one of the following clock source options can be selected for Normal Mode:

- Crystal (or ceramic resonator) or external clock at pin XTAL1 (not for XC22xxU, XC23xxS, XC27x2X, XE16xxU)
- External clock at pin CLKIN1 (not for XC22xx, XC23xx, XC27x6, XE16xx)
- Trimmed current controlled clock source (5 MHz)
  - For the remainder of this document, the term '**Trimmed current controlled clock source**' is replaced by the phrase '**internal clock**'.

The frequency of the clock source is configured by the user.

The controller's VCO is used to generate a configurable system frequency up to the controller's maximum frequency.

During the transition to Normal Operation Mode, the necessary (configurable) steps for a smooth frequency change will be applied. The SCU Driver also supports the new fractional K2 divider for the XC22xxI, XC23xxE, and XC27x8X.

The SCU Configuration Tool helps the user to find the optimum VCO configuration based on the specific parameters of the application. It also makes a proposal for the necessary frequency divider steps.

Additionally, the user can modify the system clock by calling the function **Scu_ApplyNewK2Div**.

For test purposes, the system clock can be output at a port pin via the function **Scu_OutputTestClock**.

If a crystal or ceramic resonator is used, the function **Scu_EnableHighPrecOsc** may be called at a very early stage of the initialization to reduce the waiting time for stable oscillation in the function **Scu_GoFromBaseModeToNormalMode**.

2.2 Power Saving Modes

2.2.1 Entering a Power Saving Mode

A power saving mode is entered via the function **Scu_GoFromNormalModeToPowerSavingMode**.

The parameter structure of this function specifies the details for power saving mode and wake-up.

Supported Power Saving modes

The following power-saving modes are supported:

- Normal Stopover Mode
- Stopover Mode with oscillator or external clock permanently on
- Normal Standby Mode
- Standby Mode with Fast Startup Mode (FSM), except for XC22xx, XC23xx, XC27x6, and XE16xx.

Before entering Power Saving mode

Before entering a power saving mode, ensure that:

- Any peripherals in use are switched off to save current (see "Disabling and Enabling Peripherals").
- Inputs/outputs should be brought to the state with lowest current consumption.
- Interrupts should be disabled as far as possible.
- If used as the wake-up source;
  - Reset via ESR (External Service Request) must be disabled.
  - The ESR input(s) (possibly alternate pins) must be configured accordingly
  - The ESR trap request flag(s) must be cleared. This should be done as early as possible to avoid losing a trigger.
  - The System timer must be configured appropriately.
  - The STM1 interrupt request flag must be cleared.

Wake-up from Power Saving

A wake-up from a power saving mode can be triggered by one or several events:

- Wake-Up timer (WUT)
- External Service Request (ESR) or alternate pin(s)
- System timer (STM1 interrupt trigger), (except for XC22xx, XC23xx, XC27x6, and XE16xx), and permanent external clock or wake-up oscillator clock

Wake-up Timer parameters

For the wake-up timer, additional parameters can be specified:

- Timer interval
- Auto-stop on trigger (for constant sleep time) or no auto-stop (for constant wake-up period, not XC22xx, XC23xx, XC27x6, XE16xx)
- Timer divider setting (not for XC22xx, XC23xx, XC27x6, XE16xx)

Implementing Power Saving

It is possible to use more than one power-saving mode in an application. For example, depending on the situation, Stopover Mode with clock on and FSM Standby Mode might be used.

If a power-saving mode is not required in an application, the mode can be disabled via configuration to save code and RAM.

In Standby Mode, current consumption can be further reduced by disabling the supply watchdog (SWD) or enabling the ultra-low power EVR (ULPEVR) through configuration.

When entering or exiting a power-saving mode, the internal watchdog is not served.

2.2.2 Wake-Up from Power-Saving Mode

Wake-Up from Stopover Mode

After wake-up from Stopover Mode, the SCU Driver calls a user function (**Scu_HandleStopover_Ps** with configurable name). This function executes application-defined code, initially at the configurable wake-up oscillator clock.

If a higher and/or more stable clock is required, the frequency may be increased by calling the function **Scu_UseFastClockInStopover_Ps**.

One of the following clocks can be selected for short user actions:

- 5 MHz internal PLL clock
- A configurable internal VCO clock, derived from 5 MHz internal PLL clock via VCO
  - additionally, the user can modify the system clock by calling **Scu_ApplyNewK2Div_Ps**
- A clock that is already on
  - Crystal (or ceramic resonator) clock
  - external clock at XTAL1
  - external clock at CLKIN1

Function **Scu_UseWakeupOscInStopover_Ps** must be called to reduce the clock again.

The application decides via the return value if the Stopover Mode shall be continued or not. If not, function **Scu_GoFromNormalModeToPowerSavingMode** will resume the Normal Mode.

Wake-Up from Normal Standby Mode

In case of wake-up from Normal Standby Mode, a Power Reset DMP_1 occurs. The user software has to decide which kind of reset occurred.

Wake-Up from FSM Standby Mode

In case of wake-up from FSM Standby Mode, the SCU Driver calls a user function (**Scu_HandleStandbyFsm_PsSb** with configurable name).

When the user function in FSM is started, only some important CPU registers have been set. The C startup routine has not been executed.

The user function executes application-defined code at two different speeds:

- First part running at 5 MHz internal clock
- Second part using the configurable wake-up oscillator clock
  - For this second part, the application must call **Scu_UseWakeupOscInStandbyFsm**

The application decides via the return value of the user function if the FSM Standby Mode shall be continued or not.

On exit from FSM Standby Mode, an immediate wake-up from Normal Standby Mode and a Power Reset DMP_1 will be performed.

Multiple Wake-up Sources

If multiple wake-up sources are enabled, the wake-up source can be determined by function **Scu_GetWakeupSrc**.

The wake-up source request should be cleared by function **Scu_ClearWakeupSrc**.

2.2.3 Disabling and Enabling Peripherals

Stopover Mode

In general, it is not recommended to switch off a peripheral by clearing bit MODEN in its KSCCFG / KSCFG register before entering Stopover Mode. This is because the peripheral might be in an undefined state when MODEN is set again after a wake-up.

If a peripheral is never used during cyclic wake-up, it should be switched off in Clock-off Mode via bit field COMCFG, to save power. This can be done when the peripheral is enabled for the first time by setting the MODEN bit in its KSCCFG / KSCFG register.

When entering Stopover Mode, the SCU Driver selects Clock-off Mode; on final exit from Stopover, it returns to Normal System Mode.

If a peripheral shall be used during cyclic wake-up, one of the following methods can be applied:

- The peripheral is not switched off in Clock-off Mode via COMCFG (this is the default state for many peripherals). During cyclic wake-up, the peripheral is on and can be used.
- The peripheral is switched off in Clock-off Mode via COMCFG. If the peripheral shall be used during cyclic wake-up, it is activated by a user-defined function similar to Scu_lRequestSystemMode which is called twice; first to set Normal System Mode and enable all peripherals prepared in that way, and then to set Clock-off Mode again. Note that this function must be executed from PSRAM (see section "Location of Code for Power Saving Modes"), and it should be able to handle errors. Peripherals that shall never be turned on during cyclic wake-up can be disabled via the NOMCFG bit field in their KSCCFG / KSCFG register.

Standby and FSM Standby Mode

It is not necessary to disable a peripheral in the DMP_1 domain before entering Normal Standby or FSM Standby Mode, as the DMP_1 domain is switched off anyway in these modes. On a wake-up, the peripherals in this domain will be reset.

If a peripheral in the DMP_1 domain shall be used after wake-up in any Standby Mode, it has to be enabled by writing to its KSCCFG / KSCFG register. Note that after wake-up in FSM Standby Mode the chip is in Clock-off Mode.

2.2.4 Location of Code for Power Saving Modes

For power-saving modes, parts of the SCU Driver code and of the application code must be executed from PSRAM. These code areas must be located in special areas via compiler and/or linker directives, and they have to be copied from Flash to PSRAM via the **Scu_CopyWords** function.

If interrupts or traps may occur during code execution from PSRAM, the corresponding vectors must be copied from Flash to PSRAM using the driver function **Scu_CopyVectorToPsram**.

In case of FSM Standby Mode, additional parts of SCU Driver code and application code must be stored in SBRAM and executed from PSRAM. Again, these code areas must be located in special areas via compiler and/or linker directives, and they have to be written from Flash to SBRAM via the driver function **Scu_WriteToSbram**.

2.3 Error Handling

To increase robustness, the SCU Driver performs many hardware status checks.

If a problem is detected, an error code is returned. In case of PSRAM execution, a user function (**Scu_HandleError_Ps** with configurable name) is called. The user can decide how to handle the error (e.g. perform a reset).

2.4 Delays and Timeouts

Delays or timeouts are often required. The following methods are used to meet these requirements:

Short Delays and Timeouts

- For short delays:
  - the SCU Driver inserts a series of NOPs
- For short timeouts:
  - The SCU Driver uses a polling algorithm which checks a condition several times.
  - The algorithm has a well-defined minimum number of instruction cycles before the final check is performed.
  - As soon as the condition is met, the program continues.
  - If the condition is not met after the final check, the program exits with an error.

Longer Delays and Timeouts

Software loops can be used for this purpose, but these have some disadvantages:

- Dependency on compiler/optimization, or assembler programming is necessary
- Dependency on program memory used and its alignment
- No useful actions are possible during delay or timeout

Instead, the SCU Driver uses the CCU6 timer T13.

A timeout or delay given in time units must be converted to cycles, taking into account the actual system frequency (including tolerances). To save code and execution time, the driver avoids conversion during runtime.

Function **Scu_InitTimer** configures CCU6 timer T13 as the driver timer before SCU Driver functions with delays or timeouts are required; previous CCU6 SFR contents may be saved. Function **Scu_RestoreTimer** can restore those registers after SCU Driver usage.

2.5 Configuration Concept

The SCU Driver avoids function parameters and runtime calculations wherever possible. Instead, the software makes use of simple #define values or derived values that are calculated off-line by the compiler. This has the following advantages:

- No extra code for parameter transfer, calculations, checks, or jumps
- No runtime for extra code
- Easy additional off-line calculations; e.g. for delay time cycles
- Results can be used by the customer for off-line calculations; for example for baud rate

The configuration file Scu_Cfg.h contains these #define values. This file can be generated via the SCU Configuration Tool from the original configuration file Scu_Cfg_Orig.h.

The user can make changes in the Scu_Cfg.h file itself, but this should only be required in rare cases.

To minimize the probability of configuration errors, the compiler checks some important constants that are used by the driver.

3 Configuration Parameters

The configuration of the driver for the hardware and application requirements is via the file Scu_Cfg.h.

This file can be generated via the SCU Configuration Tool ScuConfigTool from the original configuration file Scu_Cfg_Orig.h.

Most parameters are set directly by this tool, although in some rare cases special parameters must be changed manually.

No typecasts or suffixes should be used for the configured values.

The following figure shows a screenshot of the SCU configuration tool.

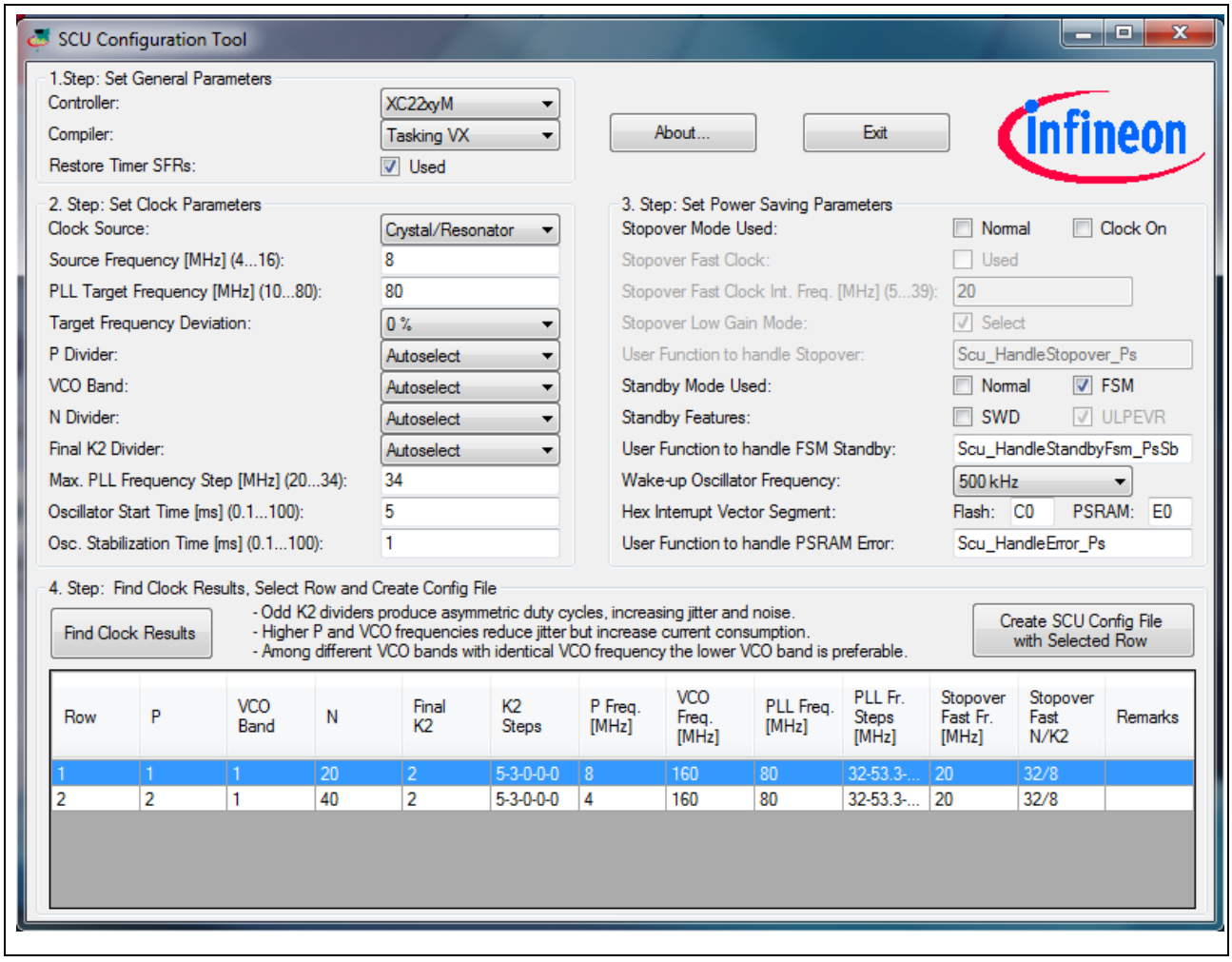

Figure 1 SCU Configuration Tool

3.1 SCU Configuration Tool

3.1.1 Setup and Use

The SCU Configuration Tool requires the Microsoft .NET framework, at least version 3.5 (client profile).

After starting the program, it is recommended to check and change the parameters in the following sequence:

- 1. Step: Set General Parameters
- 2. Step: Set Clock Parameters
- 3. Step: Set Power Saving Parameters
  - Within a parameter group, it is best to proceed from top to bottom.
  - As far as possible, the default selection should be kept (“Autoselect”).
- 4. Step: Find Clock Results, Select Row and Create Configuration File

When the button “Find Clock Results” is pressed, the clock results will be calculated and displayed in a list.

The preferred row should be selected, observing the rules at the bottom of the window.

Finally, a configuration file Scu_Cfg.h can be generated from the existing original configuration file Scu_Cfg_Orig.h by pressing the button “Create SCU Configuration File with Selected Row”.

3.1.2 Calculation of Clock Parameters

The following equation determines the PLL frequency for VCO operation:

fPLL = fR / P * N / K2INT * (12 - K2FRAC) / 12

Where:

- fR is clock source frequency (reference frequency)
- P is P divider
- N is N divider
- K2INT is integral K2 divider (K2DIV + 1)
- K2FRAC is fractional K2 divider for further fPLL reduction (XC22xxI, XC23xxE, and XC27x8X only)
  - should be 0 for other XC2000 controllers
- fR / P is frequency after P divider
- fR / P * N is VCO frequency
- fPLL is PLL output frequency = system frequency fSYS

The SCU Configuration Tool will help the user to find the optimum VCO configuration for Normal Operation Mode, based on the specific parameters of the application.

The final actual value of PLL frequency fPLL = fSYS in [Hz] for Normal Operation Mode is available as SCU_F_PLL in Scu.h and can be used by the application, for example for baud rate calculation.

Calculation of additional K2 Steps

To change the system frequency smoothly, the SCU Configuration Tool tries to find up to five intermediate K2 values to step up from the initial frequency (5 MHz) to the final PLL frequency. To step down, these values are used in the reverse order.

The difference between the PLL frequency of the new step and the previous PLL frequency should not be bigger than the maximum PLL frequency step.

If the final PLL frequency is within that range, no further intermediate step is required. All remaining steps are held at 0 and the loop can be left.

If this is not the case, an ideal real number for the next K2 step is calculated. Only if this value, when truncated to an integer, is identical to the real number, can it be used as the next step. Otherwise, one of the following two cases is handled:

- Fractional Divider is possible
- No Fractional Divider is possible

Fractional Divider is possible

First the default results for the truncated and incremented K2 divider (which is on the safe side) are calculated, without using a fractional divider.

Then it is checked if a smaller integral divider together with a fractional divider 1...3 gives a better (higher) frequency within the allowed frequency range. Among the fractional divider values with identical integral divider, the smallest fractional divider is always the best one.

Decrementing the integral divider only makes sense as long as this decremented value, together with a fractional divider of 3, yields a frequency within the allowed frequency range.

It is assumed that an intermediate K2 step with or without fractional divider can always be found.

No fractional Divider is possible

Normally, the truncated and incremented K2 divider which is on the safe side is used for the next step. However, it may happen that there is no progress in PLL frequency from the previous value because the required PLL frequency step is bigger than allowed. In this case, the truncated K2 divider is used; only if this is the step to the final PLL frequency, the K2 divider will be held at 0. A marker is set in case of too big a step.

The following figure depicts the frequency and divider values that are used to find the next K2 value. The K2 next step can then be calculated in the loop, based on the actual values.

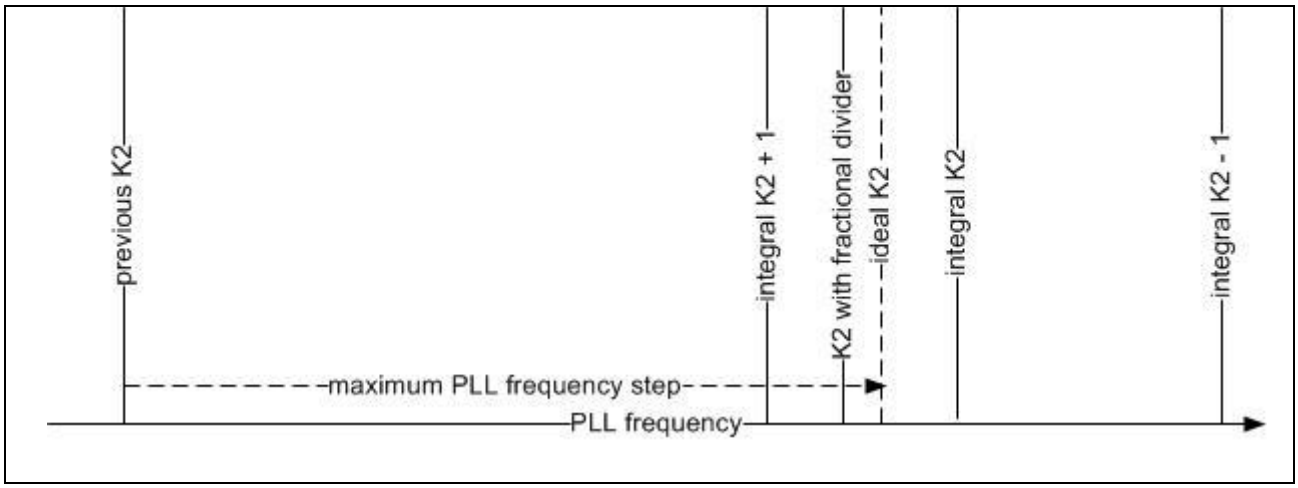

Figure 2 Finding the next K2 Value

4 External Service Request (ESR) Pins

This chapter covers the functionality of the External Service Request (ESR) features.

The ESR pins serve as multi-functional pins for a variety of different options:

- Reset trigger input
- Reset output
- Trap input
- Wake-up trigger from a power saving mode
- Trigger input for the GSC
- Overlay with other product functions
- Independent pad configuration

4.1 General Operation

Each ESR pin is equipped with an edge detection that allows the selection of the edges to be used as triggers. One, both, or no edge can be selected. Each ESRx pin can be individually configured.

Edge selection is made via the following bit fields:

- **ESRCFGx.AEDCON** if no clock is active in the power domain **DMP_M**
- **ESRCFGx.SEDCON** if a clock is active in the power domain **DMP_M**

In addition, there is a Digital (3-stage median) Filter (DF) to suppress spikes. The signal at the ESRx pin has to be held at the active signal level for at least 2 system clock cycles (*f*SYS) in order to generate a trigger. If the power domain DMP_M is not clocked, then the filter is not taken into account. The digital filter can be disabled by clearing bit **ESRCFGx.DFEN**.

Note that if an ESR trigger is generated, triggers are also generated for reset, trap, PSC, GSC, and non SCU module functions, as appropriate. To stop such triggers being generated, they need to be individually disabled in the given feature. Pins to be used as trigger input for an ESR operation have to be configured as input pins.

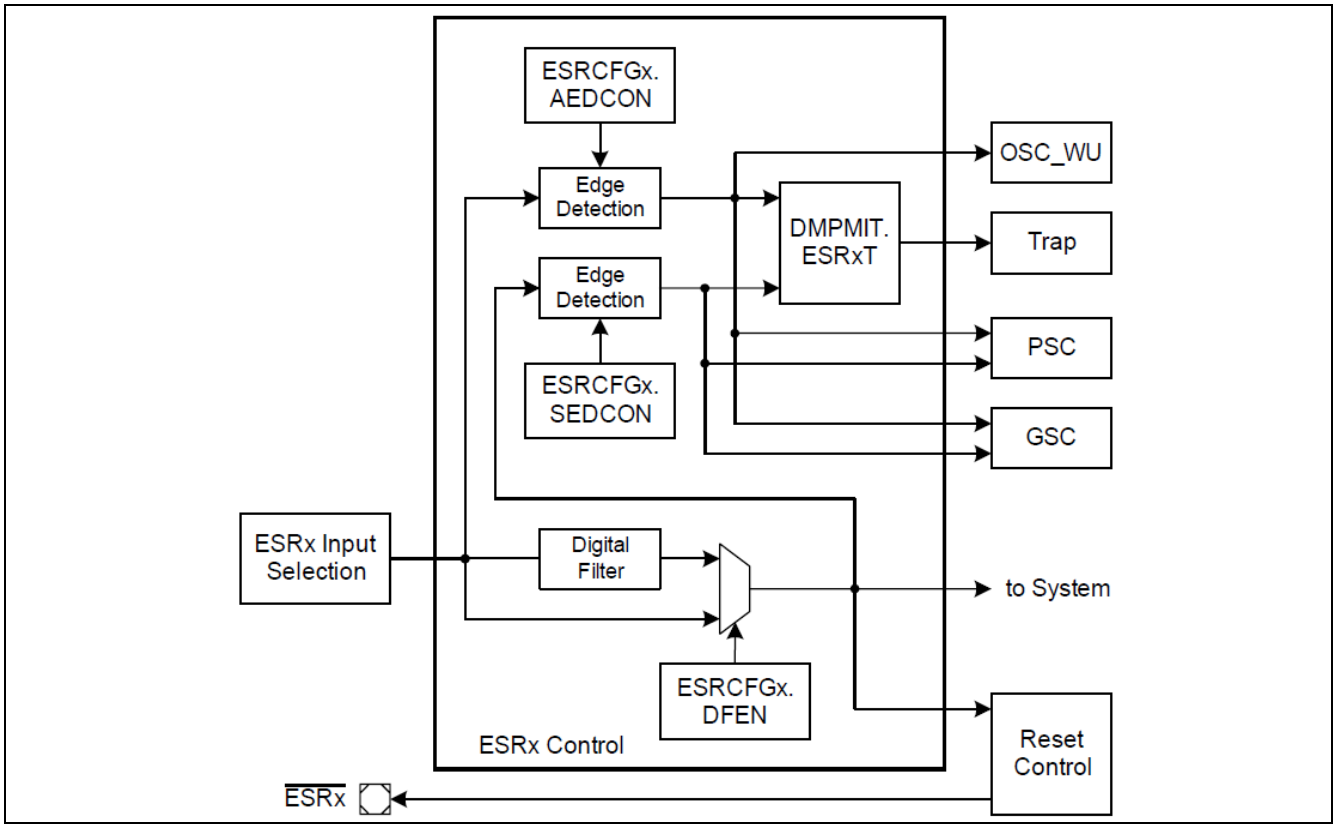

Figure 3 /ESR Control

An overlay of serial interface inputs can be configured to trigger ESR operations via the register **ESREXCONx**. Trigger generation is determined by joining the inputs (logically AND). Therefore if more than one pin is to be used for ESR trigger generation, any signal at the respective pin must have an inactive high level. It is also possible to invert some inputs to support active high levels.

To extend the overlay possibilities, the conjugated inputs of the ESRx input selection structure are combined to one common event in a second AND gate level with the ESR input stage, and the output of the combined inputs of the other input conjugation block, **ESREXCONx.ESRIN12EN**, if enabled. This allows all possible inputs to trigger an ESR function even if the second ESR logic is used for other purposes.

*Note: Pin **ESR0** does not offer an overlay with other product functions.*

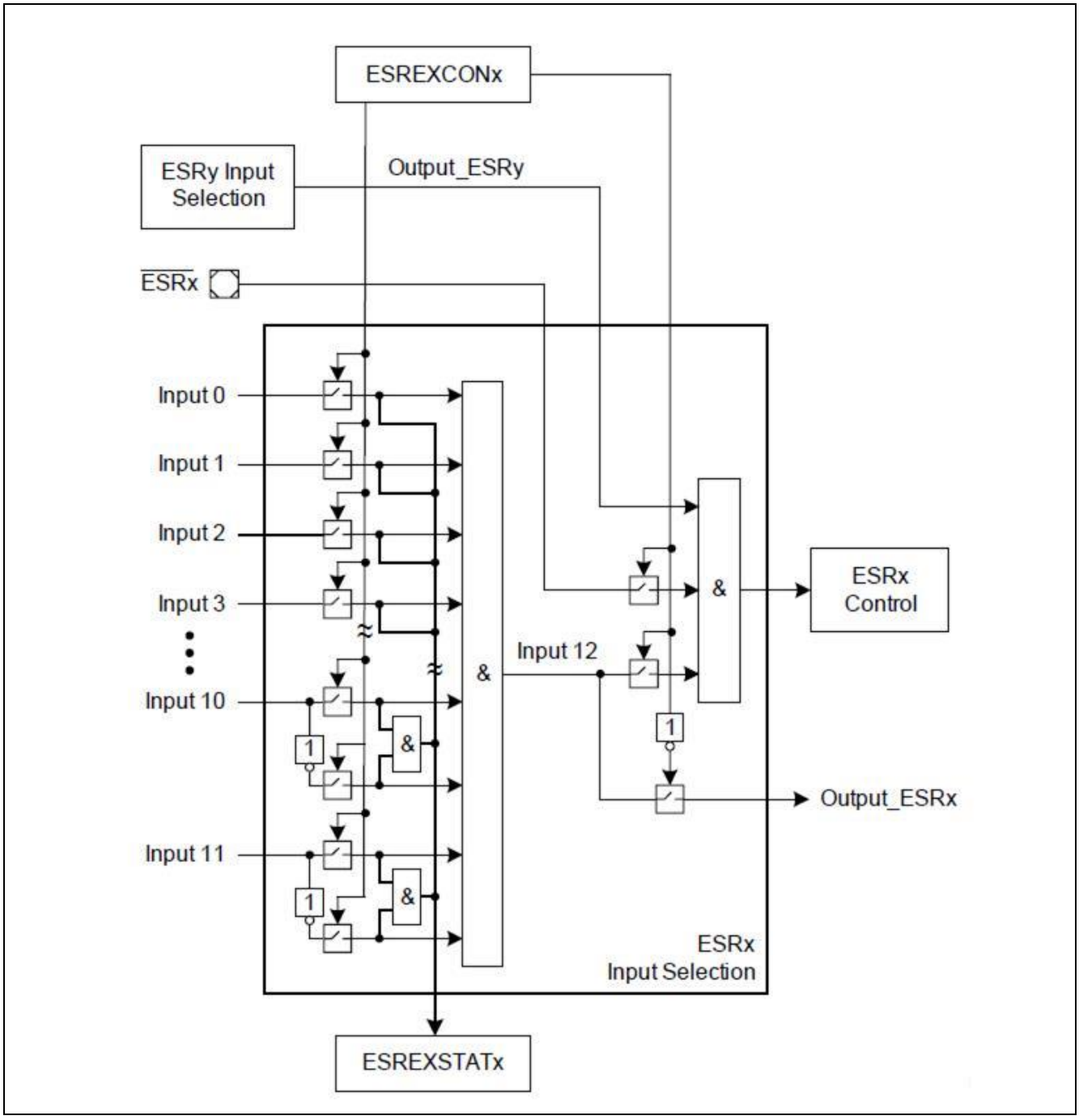

Figure 4 /ESR Input Selection

Up to three ESR pins (**ESR0**, **ESR1**, and **ESR2**) are available.

*Note: The availability of pins **ESR1** and **ESR2** is device and package dependent, and is detailed in the appropriate device data sheet.*

Even if pins **ESR1** or **ESR2** are not available in the given device, an overlay with inputs of serial interfaces can still be configured to trigger ESR operations via the registers **ESREXCON1** and **ESREXCON2**.

Other port inputs for serial communication input can also be used to generate ESR operations.

For more information about which peripheral input is on an ESR overlay pin, please refer to the appropriate User Manual.

This ESR trigger feature can be used for a variety of different uses, such as:

- Wake-up from a power saving mode on an external Interrupt or CCU6x trigger, and on a CAN or USIC operation
- Wake-up from a Clock-off Mode on an external Interrupt or CCU6x trigger, and on a CAN or USIC operation
- Request to enter a Clock-off Mode on an external Interrupt or CCU6x trigger, and on a CAN or USIC operation
- Stop input for the CCU6x modules on an external event

4.2 Implementation

This implementation description covers the following devices:

- XC22xxM
- XC23xxA
- XC27x5X
- XE16xxM

Note: For all other devices please refer to the appropriate data sheet and User Manual.

For pins **ESR1** and **ESR2**, an overlay is possible with the ESRx inputs listed in **Table 1** and **Table 2**.

Even if an ESRx pin is not available, an overlay with the ESRx inputs listed in the tables is possible.

The ESRx logic part is fully functional. Input from ESRx pins which are not available is tied to a fixed value. The default is '1', but the value can be defined in **ESRCFGx.PC**.

Table 1 ESR1 Input Connection

Input	Connected to
Input 0	Port 2.4
Input 1	Port 3.0
Input 2	Port 10.0
Input 3	Port 1.0
Input 4	Port 1.2
Input 5	Port 2.1
Input 6	Port 6.1
Input 7	Port 11.0
Input 8	Port 4.1
Input 9	Port 10.4
Input 10	Port 2.5
Input 11	Port 0.0

Table 2 ESR2 Input Connection

Input	Connected to
Input 0	Port 2.3
Input 1	Port 7.0
Input 2	Port 10.14
Input 3	Port 1.1
Input 4	Port 1.3
Input 5	Port 2.2
Input 6	Port 2.6
Input 7	Port 2.7
Input 8	Port 0.4
Input 9	XTAL 1
Input 10	Port 4.5
Input 11	Reserved, not connected

Registers for Serial Communication Overlay

The following registers needs to be configured in order to select an overlay function for serial communication:

- /ESR Configuration Register **ESRCFGx**
- /ESR External Control Register **ESREXCONx**
- Port Input/Output Control Register **PyIOCRx**

Please refer to the User Manual for detailed descriptions of these registers.

5 Application Example

5.1 Functional Description

This application example is intended to demonstrate a typical application use-case.

To reduce the overall power consumption the microcontroller is driven in the Fast Startup Mode (FSM). The port 10.0 (for example USIC0 Channel0, RxD) is used to wake-up the microcontroller when a falling edge occurs. The falling edge can be generated by a LIN message.

Because the port logic is switched off in FSM mode, it is recommended that the wakeup pin is tied to VDD via a resistor.

- Microcontroller: XC2287M
- Normal Mode: Fcpu = 80 MHz
- Fast Start up Mode: Fcpu = 5 MHz
- Power supply current in standby mode (FSM) typically 45µA…130 µA (but dependent on voltage level and µC derivate)

5.2 Configuration of the example

Wake-up from a power saving mode via RxD (LIN protocol) upon a falling edge:

- Configure ESR1 pin
  - Input no pull
  - Disable digital filter
  - No edge detection on synchronous path
  - Both edges detected on asynchronous path
- Enable ESR1 pin
- Enable the alternative input functionality
- Enable alternative input 2 (pin 10.0)
- Configure P10.0 as input

```
SCU_ESRCFG1 =  (2U   <<0U)
                | (0U <<4U)
                | (0U <<7U)
                | (2U <<9U);
SCU_ESREXCON1 = 0x2009;
P10_IOCR00     =  0x0000;
```

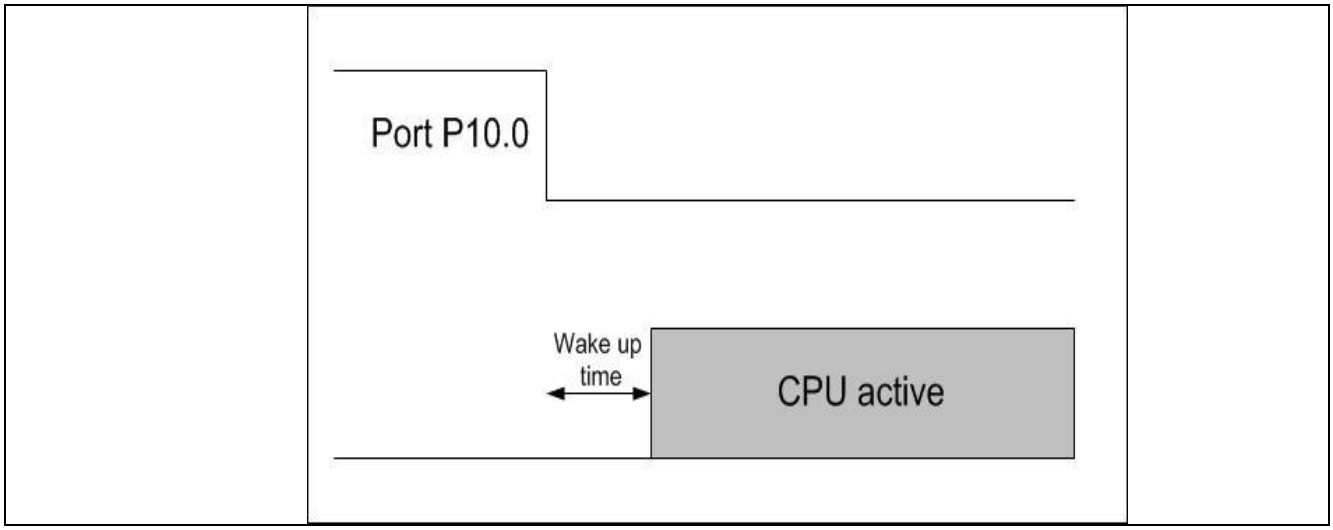

Figure 5 Application Scenario in Fast Startup Mode (FSM)

5.2.1 Flow chart

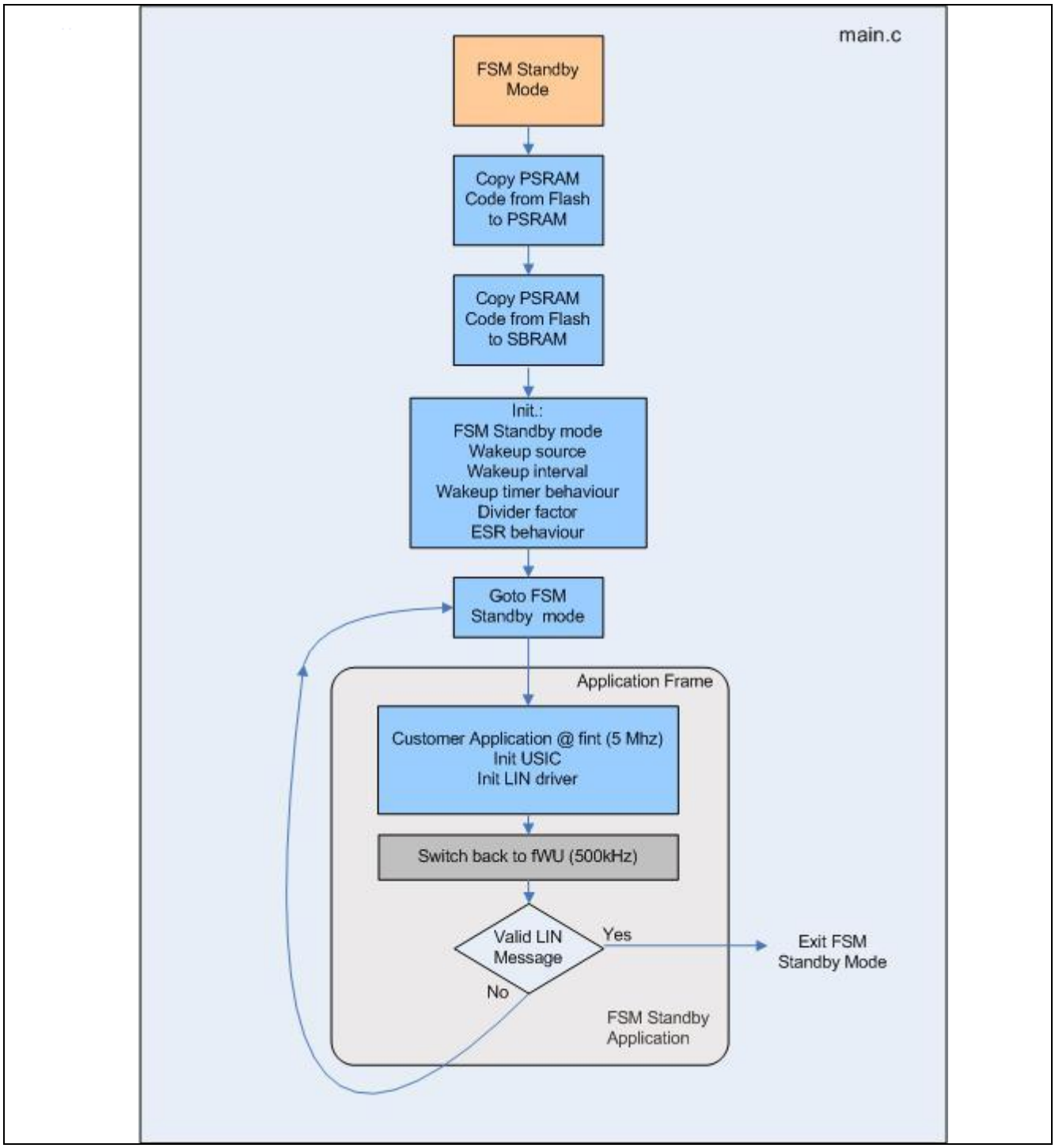

Figure 6 Flow Chart of the Application Example

5.2.2 SCU configuration header

```
/* controller selection
#define SCU_CONTROLLER SCU_CONTROLLER_BASE_LINE
/* compiler selection
#define SCU_COMPILER SCU_COMPILER_TASKING_VX
/* mandatory clock parameters
/* source for clock generation
#define SCU_CLOCK SCU_CLOCK_CRYSTAL
/* reference frequency fR in [Hz], depending on clock generation:
#define SCU_F_R 8000000
/* target value of PLL frequency fPLL in [Hz] for normal operation mode
#define SCU_F_PLL_TARGET 80000000
/* mandatory power saving parameters
#define SCU_STOPOVER_NORMAL_USED 0
#define SCU_STOPOVER_CRYSTAL_ON_USED 0
#define SCU_STANDBY_NORMAL_USED0
#define SCU_STANDBY_FSM_USED 1
/* use fast clock in stop-over mode
 /* range: 1 = fast clock can be used in stop-over mode
 /*       or
 /*       0 = fast clock cannot be used in stop-over mode
#define SCU_STOPOVER_FAST_CLOCK_USED 0
/* use restoring of driver timer SFRs
/ * range: 1 = driver timer SFRs are restored
 /*       or
 /*       0 = driver timer SFRs are not restored
#define SCU_RESTORE_TIMER_USED 1
/* wake-up oscillator frequency fWU
/* range: 140000, 180000, 270000, 500000 in [Hz]
#define SCU_F_WU 500000
/* supply watch-dog (SWD) is on/off in normal and FSM standby mode
 * range: 1 = SWD on in standby mode
 *       or
 *       0 = SWD off in standby mode
 *#define SCU_SWD_IN_STANDBY 0
/* ultra low power EVR (ULPEVR) is on/off in normal and FSM standby mode
 * range: 1 = ULPEVR on in standby mode
 *       or
 *       0 = ULPEVR off in standby mode
#define SCU_ULPEVR_IN_STANDBY 1
/* vector segment number for code execution from flash
/* range: valid segment in flash, should be consistent with compiler options
#define SCU_VECSEG_FLASH 0xC0
/* vector segment number for code execution from PSRAM
/* range: valid segment in PSRAM, should be consistent with compiler options
#define SCU_VECSEG_PSRAM 0xE0
/* user function to handle error in PSRAM
 /* range: valid user function
#define SCU_HANDLE_ERROR_PS Scu_HandleError_Ps
/* user function to handle stop-over mode
/* range: valid user function
#define SCU_HANDLE_STOPOVER_PS Scu_HandleStopover_Ps
/* mask for flash busy in SCU_IMB_FSR_BUSY register
/* range: 0x1...0x7F
#define SCU_MASK_IMB_FSR_BUSY_ALL 0xF
```

Figure 7 Scu_Cfg.h Configuration

The following figure shows the application code required in the **main.c** file, and illustrates the part that is executed after wakeup.

```
unsigned int Scu_HandleStandbyFsm_PsSb(void)
{
  unsigned int WakeupSrc;


  /*
   * place time-critical actions here (fSYS = 5 MHz): *************************
   */
  /* get wake-up source */
  WakeupSrc = Scu_GetWakeupSrc();

  /* switch from PLL oscillator to wake-up oscillator (necessary) */
  Scu_UseWakeupOscInStandbyFsm();

  /****structure example for wakeup decision with LIN************
   *
   * Initialize LIN
   *
   * Receive LIN Message
   *
   *    if (valid_LIN_Message)
   *    {
   *            return (0);             //Exit FSM Mode
   *    }
   *    else
   *    {
   *            return (1U);    //Continue FSM Mode
   *    }
   ************************************************************/

          Test_ClearLedPin(5);
          Scu_OutputTestClock(SCU_EXTCLK_F_SYS, SCU_EXTCLK_P28);
          while(1);


} /* end of function Scu_HandleStandbyFsm_PsSb */

/* end of code to be executed from PSRAM, saved in SBRAM */
#if(SCU_COMPILER == SCU_COMPILER_TASKING_VX)
#  pragma section default
#elif(SCU_COMPILER == SCU_COMPILER_TASKING_CLASSIC)
#  pragma default_attributes
#endif

#endif /* (SCU_STANDBY_FSM_USED) */

#endif /* (SCU_STANDBY_FSM_USED || SCU_STANDBY_NORMAL_USED \
|| SCU_STOPOVER_CLOCK_ON_USED || SCU_STOPOVER_NORMAL_USED) */
```

Figure 8 Function Scu_HandleStandbyFsm_PsSb (Parts of main.c)

6 Limitations and Assumptions

6.1 Limitations

6.1.1 Possible Configuration Issues

The SCU Driver does not perform a full check of the user configuration in Scu_Cfg.h.

By using the SCU Configuration Tool, many configuration problems can be avoided. However, the settings offered by the SCU Configuration Tool are mainly based on the controller used. Certain derivatives may be specified in a more restrictive way.

Known issues are:

- Not all XC22xx, XC23xx, XC27x6, and XE16xx are specified for Normal and FSM Standby Mode.
- Not all XC22xxU, XC23xxS, XC27x2X and XE16xxU are specified for high PLL frequencies of up to 80 MHz.
- Not all XC22xxL, XC23xxD, XC27x3X, and XE16xxL are specified for high PLL frequencies of up to 66 MHz.

Please refer to the data sheet of the respective controller.

6.1.2 Possible Problems with User-written PSRAM Programs

When the Flash is switched off, user-written code located in PSRAM may cause problems due to one of the following reasons:

- Library Functions
- Constants
- Switch Statements

Note: The SCU Driver itself will not cause any of these problems.

Library Functions

Libraries are typically located in Flash. Therefore, any call to the library will go wrong. This is also true for operations that need the runtime library (long division for example).

Constants

Constants (defined via "const", "#pragma romdata", or string literals) are typically located in Flash and are not automatically copied to PSRAM. This can be solved in one of the following ways:

- Constants are located in a special part in Flash which is copied during runtime like the PSRAM code, using **Scu_CopyWords**.
- Via a special compiler keyword, for example “#pragma ramdata” for Tasking VX, constants are located in RAM and are copied from Flash to RAM during startup.

Switch Statements

Depending on the optimization settings and the user program, a jump table located in Flash may be generated. This can be avoided in one of the following ways:

- The switch statement is replaced by if - else if - ... – else.

- Via a special compiler keyword, for example “pragma linear_switch” for Tasking VX, a linear switch can be forced; for Tasking VX, this pragma is used in Scu.h.

6.1.3 Possible Problems with User-written FSM Standby Programs

As the Flash is switched off in FSM Standby Mode, the issues listed in section "Possible Problems with User-written PSRAM Programs" may occur.

Please note also that the C startup routine is not executed in FSM Standby Mode. Only very important initializations are performed, such as:

- Disabling of watchdog timer
- Disabling of functional resets
- Disabling of interrupts
- Setting of first three data page pointers
- Disabling of EBC mode to enable normal use of EBC pins

Additional initializations must be performed by the user if required, for user stack or interrupts / traps for example.

Note: The SCU Driver itself will not need additional initializations.

6.1.4 Clock Issues

No Special Handling of External Clock at Crystal Input

The SCU Driver software handles an external clock connected to the XTAL1 input like a high precision oscillator with crystal or ceramic resonator. For example, there is no different setting of the HPOSCCON register, and the waiting time for high precision oscillator start is not removed.

Disadvantages of Fast Clock in Stopover Mode

- There is no emergency handling for the crystal (or ceramic resonator) or an external clock.
- The clock is switched directly from 5 MHz to the crystal (or ceramic resonator) or external clock frequency and back. This might be a problem if the frequency is high.
- If the crystal (or ceramic resonator) or the external clock frequency is 5 MHz or lower, the clock might be switched directly, without using the internal PLL oscillator.

6.1.5 Tool Chain Issues

Tasking VX Tool Chain

Some microcontrollers are not correctly supported by older tool chains (VX V2.5r1 and earlier):

- XC22xxL, XC23xxD, XC27x3X, XE16xxL
- XC22xxU, XC23xxS, XC27x2X, XE16xxU
- XC22xxI, XC23xxE, XC27x8X

As a workaround, an older controller type can be selected in the compiler settings.

Major SFR problems are known with older compiler versions, for example Tasking C166 VX V2.3 r1 and earlier. SFR problems are also known with newer versions.

With Tasking C166 VX, some compiler optimization techniques should be disabled with SCU related code.

Tasking Classic Tool Chain

Some microcontrollers are not correctly supported by the newer tool chains (Tasking C166 Classic V8.8r1 and earlier):

- XC22xxL, XC23xxD, XC27x3X, XE16xxL
- XC22xxU, XC23xxS, XC27x2X, XE16xxU
- XC22xxI, XC23xxE, XC27x8X

For Tasking C166 Classic V8.8r1, updated SFR files are available. As a workaround, an older controller type or a user-defined controller can be selected in the compiler settings.

Major SFR problems are known with older compiler versions, such as V8.7 r2 and earlier.

Other Compiler Issues

With Keil compiler V7.04, the following issues are known:

- Power-saving features are not supported
- The tool chain will complain about unused static functions

6.1.6 MISRA Compliance Issues

The SCU Driver software has been tested for MISRA 2004 compliance with PC-lint V9.00h and the corresponding option file au-misra2.lnt.

Please consider the following remarks:

- Some MISRA 2004 exceptions have been added to the SCU project files.
- MISRA 2004 violations related to compiler specific language extensions must be suppressed in an option file.
- MISRA 2004 violations related to compiler header files must be suppressed in an option file.
- Only PC-lint messages that are MISRA 2004 relevant have been taken into account. Other messages can be suppressed in an option file.

However, this MISRA test is not fully sufficient. Certain rules are not supported at all, or are only partially supported by the tool at the moment.

7 Conclusion

The SCU Driver offers software that supports the customer to configure the clock system and the power-down modes.

The XC2000 family, together with the SCU unit, supports several powerful mechanisms to address different application scenarios.

Following the ‘family’ concept, Infineon is able to offer a wide range of different devices to meet the wide diversity of requirements in the market today and in the future.

Active Modes

- Normal Mode
- Internal Clock Mode
- Direct Drive Mode

Power Down modes

- Stopover Mode
- Standby Mode (Fast Startup Mode)
- Standby Mode

Driver Optimization

The Driver has been optimized in terms of:

- Robust design
- Error management

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG