Never stop thinking

XC2000 Family

AP16168

Clock Generation and Power Management
SCU Driver Introduction

Application Note

V1.0 2009-09

Microcontrollers

Edition 2009-09

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.

Device1

Revision History: V1.0, 2009-09

Previous Version: none

Page	Subjects (major changes since last revision)

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 this document.
Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents

1 Functional Overview

Infineon provides software examples that enable the customer to configure and use the XC2000 System Control Unit (SCU) according to their needs, without any detailed knowledge of this powerful and complex unit.

The SCU Driver offers API functions that can be used for the following main purposes:

- Control of system clock (Normal Mode)
- Control of power-saving modes

There are several types of API functions:

- Global functions are needed for typical applications.
- Private functions are used internally by the driver. They are documented for reference purposes. The user may copy and paste these functions for special purposes.
- Test functions help the user to get their application running.

Features of the functions are described in the following sections.

1.1 Normal Mode

After reset, the system will enter 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 external clock at crystal input
- External clock at pin CLKIN1
- Trimmed current controlled clock source (5 MHz)

*Note: For the remainder of this Application note, 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 clock of up to 80 MHz.

The system clock can be output at port pin 2.8 for test purposes.

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

1.2 Power Saving Modes

1.2.1 Entering a Power Saving Mode

A power-saving mode is entered via SCU Driver function **Scu_GoFromNormalModeToPowerSavingMode**.

The parameter structure of this function specifies the details for power-saving mode and wake-up. The following power-saving modes are supported:

- Normal Stop-Over Mode
- Stop-Over Mode with crystal oscillator permanently on
- Normal Standby Mode
- Standby Mode with Fast Startup Mode (FSM)

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

- Wake-Up timer (WUT)
- External Service Request (ESR) pin(s)
- Alternative ESR pins (CAN, SPI, LIN)

For the timer, additional parameters can be specified:

- Timer interval
- Auto-stop on trigger or no auto-stop (for constant sleep time or constant wake-up period)
- Timer divider setting

Before entering a power-saving mode, the user should take care of the following items:

- Used peripherals should be switched off to save current
- Inputs/outputs should be brought to the state with lowest current consumption
- Interrupts should be disabled

If a power-saving mode is not used at all in an application, the mode can be disabled via configuration to save code. In Standby Mode, current consumption can further be reduced by disabling the supply watch-dog by configuration.

1.2.2 Wake-Up from Power-Saving Mode

Wake-Up from Stop-Over Mode

After wake-up from Stop-Over 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 needed, the frequency may be increased by calling the function **Scu_UseFastClockInStopover_Ps**. One of the following clocks can be selected for short user actions:

- Configured and available crystal, crystal clock or clock at CLKIN1
- Configurable internal VCO clock, derived from 5 MHz internal PLL clock via VCO; additionally, the user can modify the system clock by calling function Scu_ApplyNewK2Div_Ps.
- 5 MHz internal clock

Finally, function **Scu_UseWakeupOscInStopover_Ps** must be called to reduce the clock again.

The application decides via the return value if the Stop-Over Mode shall be continued or not. If not continued, function **Scu_GoFromNormalModeToPowerSavingMode** will resume the normal mode.

Wake-Up from Standby Mode (Fast Startup Mode)

In the case of wake-up from FSM (Fast Startup Mode) Standby Mode, the SCU Driver calls a user function (**Scu_HandleStandbyFsm_PsSb** with configurable name). This 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 the 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 or shall not be continued. After exit from FSM Standby Mode, a DMP_1 reset is performed.

If peripherals are to be activated during execution of the user function, the SCU Driver function **Scu_RequestSystemMode** should be called with following parameters:

- SCU_SYSTEM_MODE_NORMAL to enable the peripherals
- SCU_SYSTEM_MODE_CLOCK_OFF to disable them again

Wake-Up from Standby Mode

A DMP_1 reset happens after a wake-up from normal Standby Mode.

Multiple Wake-Up Sources

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

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

1.2.3 Location of Code for Power Saving Modes

For power-saving modes, parts of the SCU Driver code and application code must be executed from PSRAM. These code sections must be located in special areas using compiler directives. They have to be copied from flash to PSRAM via the function **Scu_CopyWords**.

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

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

1.3 Error Handling

To make the software more robust, the SCU driver performs many hardware status checks. If any problem is detected, an error code is returned. In the case of PSRAM execution, a user function **Scu_HandleError_Ps** (with configurable name) is called.

The user can decide how to handle the error (Perform a reset for example).

1.4 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. 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. The algorithm has a well defined minimum number of instruction cycles before the final check is performed.

Longer Delays and Timeouts

Software loops can be used, but with some disadvantages:

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

Therefore, 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.

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

1.5 Configuration Concept

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

- No extra code for parameter transfer, calculations, checks, jumps
- No runtime for extra code
- Complex calculations possible; For example, for optimum size and minimum number of PLL ramp steps

- Easy additional off-line calculations; For example, for delay time cycles

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

Not all divider calculations are supported. As an alternative to the calculation by the compiler, the user can calculate the values by themselves and put the values in the **SCU_CFG.h**. Figure 1 and Figure 2 which follow, illustrate an example of how to do this manually.

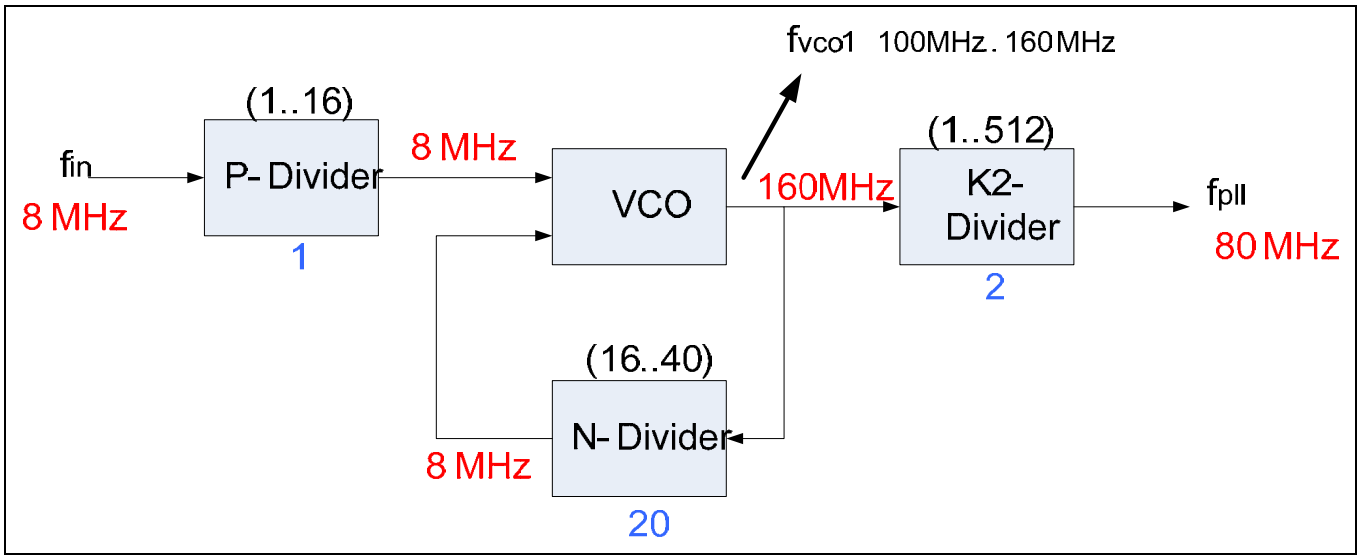

Figure 1 PLL Dividers (Example)

In Normal Mode the reference frequency f_R is divided by a factor P, multiplied by a factor N and then divided by a factor K2. The output frequency is given by:

$$f_{PLL} = \frac{N}{P \cdot K2} \cdot f_R$$

```
/* reference frequency fR in [Hz], depending on clock generation:
 * - fR = fXTAL for SCU_CLOCK_CRYSTAL
 * - fR = fINT for SCU_CLOCK_INTERNAL (typically 5000000)
 * - fR = fCLKIN1 for SCU_CLOCK_CLKIN1
 * range: 4000000...25000000
 * example: #define SCU_F_R 5400000 */
#define SCU_F_R 8000000

/* target value of PLL frequency fPLL in [Hz] for normal operation mode
 * range: 10000000...80000000
 * note: The actual frequency may be different. Its value SCU_F_PLL is available
 * in Scu.h and can be used by the application e.g. for baud rate calculation.
 * example: #define SCU_F_PLL_TARGET 40000000 */
#define SCU_F_PLL_TARGET 80000000

/* P divider for normal operation mode
 * range: 1...16, default: calculated
 * example: #define SCU_P 2 */
#define SCU_P 1
/* N divider for normal operation
 * range: 16...40, default: calculated
 * example: #define SCU_N 19 */
#define SCU_N 20
/* final K2 divider for normal operation mode
 * range: 1...512, default: calculated
 * example: #define SCU_K2 5 */
#define SCU_K2 2
```

Figure 2 SCU_CFG.h

1.6 Development Tools

The following tool chains are supported:

- Tasking C166 VX (all modes)
- Tasking C166 Classic (all modes)
- Keil C166 (Normal Mode)

There is a special configuration switch for selection of the required compiler.

2 Configuration of the Driver

The driver consists of three files and the application related main file that includes the customer software. The driver is configured with the file **Scu_CFG.h**.

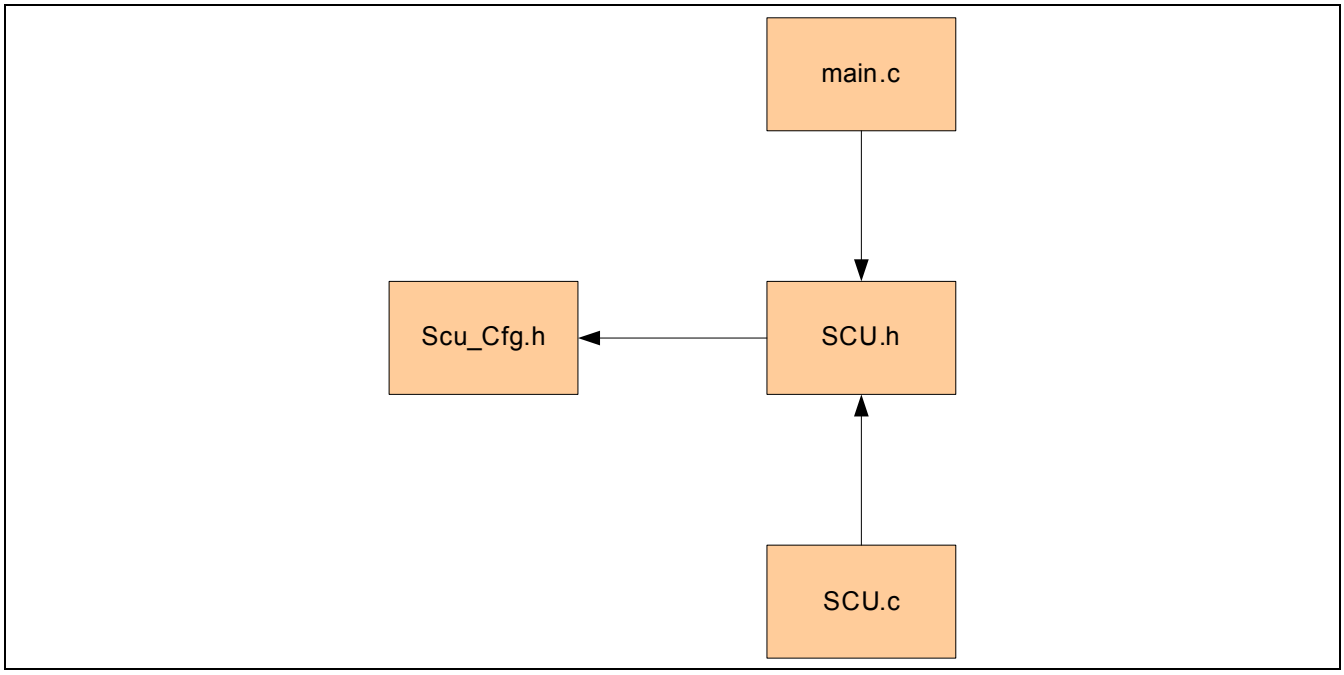

Figure 3 File Structure

Table 1 File Contents

File Name	File Contents	User Changes Possible
Scu.c	Code for SCU Driver	No
Scu.h	Header file exporting the SCU Driver functionality, for example by macros, type definitions and function prototypes	No
Scu_Cfg.h	Header file for SCU Driver configuration comprising user definitions	Yes
main.c	Code for SCU main; may be replaced by customer software	Yes

The user of the SCU Driver needs to include only the file Scu.h to import the functionality of the driver.

2.1 Normal Mode

If a reset occurs (PORST, ((internal) Application Reset) the system will start to execute the internal bootcode. After leaving the bootcode the microcontroller uses the internal clock. The CPU is clocked with 10 MHz. As part of the configuration, one of the following clock source options can be selected for normal mode:

- Crystal or external clock at crystal input
- External clock at pin CLKIN1
- Internal clock (5 MHz)

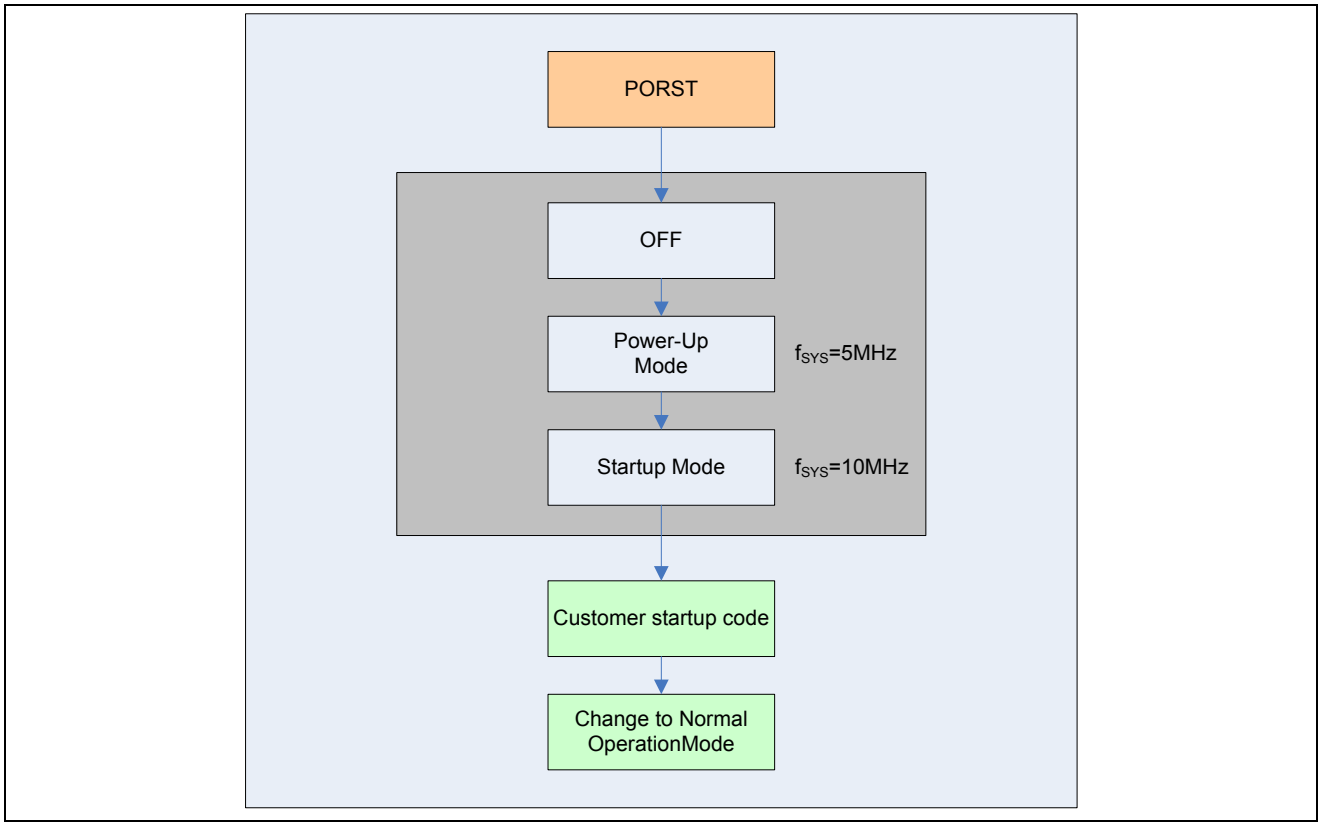

Figure 4 Flowchart Normal Mode

The configuration file **Scu_Cfg.h** is used to configure the driver with:

- Controller selection
- Compiler selection
- Source for Clock Generation
- Reference Frequency
- Target PLL Frequency
- Restoring of Driver Timer SFRs used
- Clock Parameters (optional)

For the generation of long delays the SCU Driver uses the CCU6 timer T13.

The function **Scu_RestoreTimer** can restore those registers after SCU Driver usage.

```
/* controller selection
 * range: SCU_CONTROLLER_ALPHA_LINE (e.g. XC2287)
 *        or
 *        SCU_CONTROLLER_BASE_LINE  (e.g. XC2287M)
 *        or
 *        SCU_CONTROLLER_VALUE_LINE (e.g. XC2365B)
 *        or
 *        SCU_CONTROLLER_HIGH_LINE
 * note: also compiler settings must be changed accordingly!
 * example: #define SCU_CONTROLLER SCU_CONTROLLER_BASE_LINE */
#define SCU_CONTROLLER SCU_CONTROLLER_BASE_LINE

/* compiler selection
 * range: SCU_COMPILER_TASKING_VX (e.g. Tasking C166 VX V2.3 r3)
 *        or
 *        SCU_COMPILER_TASKING_CLASSIC (e.g. Tasking C166 V8.7 r3)
 *        or
 *        SCU_COMPILER_KEIL (e.g. Keil C166 V6.14)
 * notes: - If a wrong compiler is selected, many compilation errors will occur.
 *        - No warnings should occur except with the Keil tool chain which will
 *          complain about unused static functions.
 *        - Care should be taken that the compiler's C startup program does not
 *          set SFRs handled by the SCU driver, e.g. PLLCON<x>.
 *
 * example: #define SCU_COMPILER SCU_COMPILER_KEIL */
#define SCU_COMPILER SCU_COMPILER_TASKING_VX

/* restoring of driver timer SFRs used
 * range: 1 = driver timer SFRs can be restored
 *        or
 *        0 = driver timer SFRs cannot be restored
 * note:  Restoring needs additional code and RAM.
 * example: #SCU_RESTORE_TIMER_USED 1 */
#define SCU_RESTORE_TIMER_USED 0

/*
 * mandatory clock parameters ************************************************
 */
/* source for clock generation
 * range: SCU_CLOCK_CRYSTAL (crystal or external clock at crystal input)
 *        or
 *        SCU_CLOCK_CLKIN1 (input CLKIN1 via PLL, only base line controllers or
 *        higher)
 *        or
 *        SCU_CLOCK_INTERNAL (internal PLL oscillator)
 * example: #define SCU_CLOCK SCU_CLOCK_CRYSTAL */
#define SCU_CLOCK SCU_CLOCK_CRYSTAL

/* reference frequency of clock source fR in [Hz], depending on clock:
 * - fR = fXTAL for SCU_CLOCK_CRYSTAL (e.g. 8000000 for starter kit)
 * - fR = fCLKIN1 for SCU_CLOCK_CLKIN1
 * - fR = fINT for SCU_CLOCK_INTERNAL (typically 5000000)
 * range: 4000000...25000000
 * example: #define SCU_F_R 5400000 */
#define SCU_F_R 8000000

/* target value of PLL frequency fPLL = fSYS in [Hz] for normal operation mode
 * range: 10000000...80000000
```

Figure 5 Screenshot of Scu_Cfg.h (Normal Mode)

2.1.1 Controller Selection

Table 2 Controller Selection

Name:	SCU_CONTROLLER
Description:	Selection of controller
Type:	(int)
Default Value:	None
Dependency:	None
Range:	SCU_CONTROLLER_ALPHA_LINE (e.g. XC2287) or SCU_CONTROLLER_BASE_LINE (e.g. XC2287M) or SCU_CONTROLLER_VALUE_LINE (e.g. XC2365B) or SCU_CONTROLLER_HIGH_LINE

Note: Compiler settings must be changed accordingly!

2.1.2 Compiler Selection

Table 3 Compiler Selection

Name:	SCU_COMPILER
Description:	Selection of compiler
Type:	(int)
Default Value:	None
Dependency:	None
Range:	SCU_COMPILER_TASKING_VX (e.g. Tasking C166 VX V2.3 r3) or SCU_COMPILER_TASKING_CLASSIC (e.g. Tasking C166 V8.7 r3) or SCU_COMPILER_KEIL (e.g. Keil C166 V6.14)

Note: If an invalid compiler is selected, many compilation errors will occur.

Note: No warnings should occur except with the Keil tool chain which will complain about unused static functions.

Note: Care should be taken that the compiler's C startup program does not set SFRs handled by the SCU Driver; for example, the PLLCON<x>.

2.1.3 Restoring of Driver Timer SFRs used

Table 4 Restoring of Driver Timer SFRs used

Name:	SCU_RESTORE_TIMER_USED
Description:	Restoring of driver timer SFRs used
Type:	(int)
Default Value:	None
Dependency:	None

Name:	SCU_RESTORE_TIMER_USED
Range:	1 = driver timer SFRs can be restored or 0 = driver timer SFRs cannot be restored

Note: Restoring needs additional code and RAM.

2.1.4 Source for Clock Generation

Table 5 Source for Clock Generation

Name:	SCU_CLOCK
Description:	Source for clock generation
Type:	(int)
Default Value:	None
Dependency:	None
Range:	SCU_CLOCK_CRYSTAL (crystal or external clock at crystal input) or SCU_CLOCK_CLKIN1 (input CLKIN1 via PLL, only for newer devices) or SCU_CLOCK_INTERNAL (internal PLL oscillator)

2.1.5 Reference Frequency

Table 6 Reference Frequency

Name:	SCU_F_R
Description:	Reference frequency of clock source fR in [Hz], depending on clock: fR = fXTAL for SCU_CLOCK_CRYSTAL fR = fCLKIN1 for SCU_CLOCK_CLKIN1 fR = fINT for SCU_CLOCK_INTERNAL (typically 5000000)
Type:	(long)
Default Value:	None
Dependency:	SCU_CLOCK
Range:	4000000...25000000

2.1.6 Target PLL Frequency

Table 7 Target PLL Frequency

Name:	SCU_F_PLL_TARGET
Description:	Target value of PLL frequency fPLL = fSYS in [Hz] for normal operation mode
Type:	(long)
Default Value:	None
Dependency:	None
Range:	10000000...80000000

Note: The actual frequency may be different. Its value SCU_F_PLL is available in Scu.h and can be used by the application, for baud rate calculation for example.

Note: To avoid a different actual frequency, it is possible to set the frequency after the P divider manually, or to set the P divider manually. This will increase the frequency jitter.

2.2 Stop-Over Mode

The Stop-Over mode reduces the overall current consumption to 0.7mA (typically) and allows a fast wake scenario. The system clock is switched off but the embedded voltage regulators are active and supply the internal logic. The complete system behavior is frozen. Typically the system can be activated either by a wakeup timer or with an external event (CAN, seriellel interface, or any kind of edge).

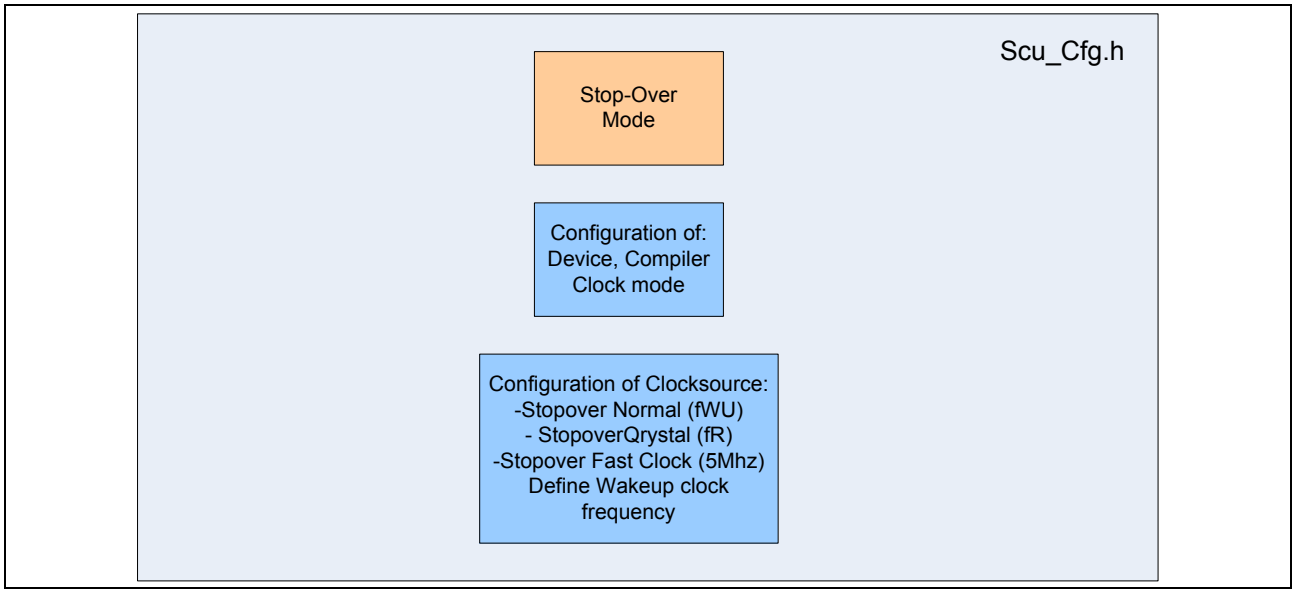

Figure 6 Configuration Flow in SCU_Cfg.h

The configuration file **Scu_Cfg.h** is used to configure the driver in Stop-Over Mode.

- Configuration of clocksource in Stop-Over Mode
  - Stopover Normal (Wakeuptimer clock)
  - Stopover Qrystal (Qrystal clock)
  - Stopover Fast Clock (internal clock)
- Wakeup clock frequency

The application specific issues are handled in the file main.c. Before entering Stop-Over mode, some initialization has to be done.

- Configuration of Stopover mode
- Configuration of Wakeup Timer
  - Source (Timer, ext. event)
  - Time Interval
  - Wakeup Behavior
  - Timer Divider
- Configuration of ESRx

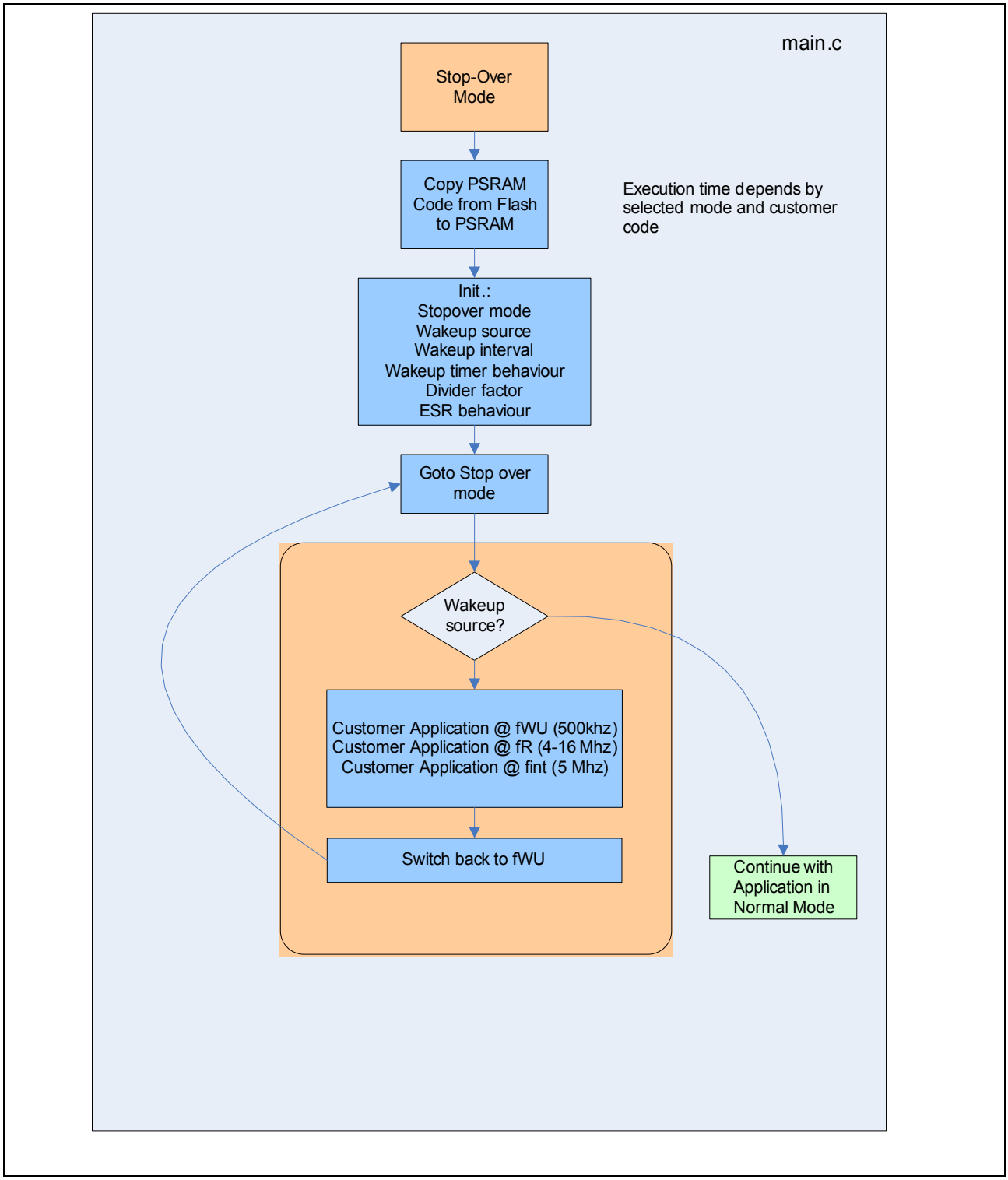

Figure 7 Stop-Over Flow in Function main

Depending on their requirements, the customer can execute the application with wakeup timer clock, crystal clock, or internal clock.

2.2.1 Normal Stop-Over Mode used

Table 8 Normal Stop-Over Mode used

Name:	SCU_STOPOVER_NORMAL_USED
Description:	Normal Stop-Over Mode used
Type:	(int)
Default Value:	None
Dependency:	None
Range:	1 = mode can be used or 0 = mode cannot be used

Note: Enabling a mode needs additional code, also in PSRAM.

Note: If all modes are disabled, no other power-saving parameters are used.

2.2.2 Stop-Over Mode with Crystal on used

Table 9 Stop-Over Mode with Crystal on used

Name:	SCU_STOPOVER_CRYSTAL_ON_USED
Description:	Stop-Over Mode with Crystal on used
Type:	(int)
Default Value:	None
Dependency:	None
Range:	1 = mode can be used or 0 = mode cannot be used

Note: See Normal Stop-Over Mode Used.

2.2.3 Fast Clock with Stop-Over used

Table 10 Fast Clock with Stop-Over used

Name:	SCU_STOPOVER_FAST_CLOCK_USED
Description:	Fast clock is used after wake-up from Stop-Over Mode
Type:	(int)
Default Value:	None
Dependency:	SCU_STOPOVER_NORMAL_USED, SCU_STOPOVER_CRYSTAL_ON_USED
Range:	1 = fast clock can be used or 0 = fast clock cannot be used

Note: Fast clock needs additional code, also in PSRAM.

Note: Option is only evaluated if any Stop-Over Mode is used.

2.2.4 K2 Divider for fast internal VCO Clock in Stop-Over Mode

Table 11 K2 Divider for fast internal VCO Clock in Stop-Over Mode

Name:	SCU_K2_FAST_CLOCK_IN_STOPOVER
Description:	K2 divider for fast internal VCO clock in Stop-Over Mode
Type:	(int)
Default Value:	None
Dependency:	SCU_STOPOVER_NORMAL_USED, SCU_STOPOVER_CLOCK_ON_USED, SCU_STOPOVER_FAST_CLOCK_USED
Range:	1...512, typically limited to ~8...32 (see notes)

Note: Option is only evaluated if fast internal VCO clock in Stop-Over Mode is used.

Note: Resulting fSYS = 160 MHz / SCU_K2_FAST_CLOCK_IN_STOPOVER

Note: As system clock depends on K2 during execution, the rules for smooth system clock frequency stepping must be applied.

2.2.5 Wake-Up Oscillator Frequency fWU

Table 12 Mask for Flash busy

Name:	Wake-up oscillator frequency fWU
Description:	Frequence for Wake-up oscillator
Type:	(int)
Default Value:	None
Dependency:	None
Range:	140000, 180000, 270000, 500000

Note: Fast clock needs additional code, also in PSRAM.

Note: Option is only evaluated if any Stop-Over Mode is used.

2.2.6 Oscillator Low Gain in Stop-Over Mode with Crystal Clock on

Table 13 Oscillator Low Gain in Stop-Over Mode with Crystal Clock on

Name:	SCU_OSC_LOW_GAIN_IN_STOPOVER_CRYSTAL_CLOCK_ON
Description:	Oscillator low gain in stop-over mode with clock on
Type:	(int)
Default Value:	None
Dependency:	SCU_STOPOVER_CLOCK_ON_USED, SCU_CLOCK_CRYSTAL
Range:	1 = oscillator low gain is on in stop-over mode with crystal clock on or 0 = oscillator low gain is off in stop-over mode with crystal clock on

Note: Enabling the oscillator low gain mode saves current.

Note: Option is only evaluated if stop-over mode with clock on and crystal is used.

2.3 Standby Mode

The Standby mode reduces the overall current consumption to 70µA (typically). The Fast Startup mode (FSM) allows a fast wake scenario. The complete system is switched off (DMP_1 = off) and only the embedded voltage regulator for the wakeup domain is active (DMP_M = on). That means that the CPU, memory, and peripheral are off. The content of the Standby SRAM is conserved and optionally the Wakeup Timer (WUT) and the Supply Watchdog (SWD) are active.

The minimum power consumption can be achieved when the WUT and SWD are switched off, the embedded voltage regulator is replaced by the Ultra Low Power embedded voltage regulator and the power supply is reduced to 3.3V.

Typically the system can be activated either by a wakeup timer or with an external event (CAN, seriellel interface, or any kind of edge). If a wake up scenario occurs a complete system boot is generated. In FSM Startup mode the microcontroller executes code from the PSRAM.

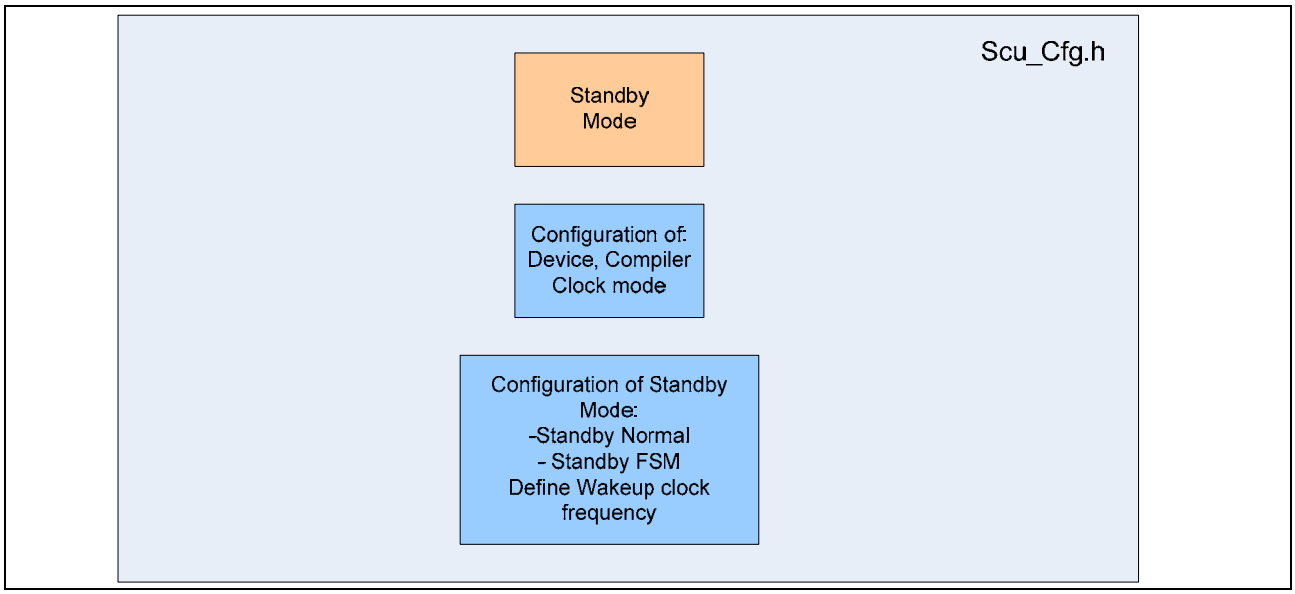

Figure 8 Configuration Flow in SCU_Cfg.h

The configuration file **Scu_Cfg.h** supports the user to configure the driver in Standby Mode.

- Configuration of clocksource in Standby Mode
  - Wakeuptimer clock
  - Internal clock
- Wakeup clock frequency
- SWD configuration

The application specific issues are handled in the file main.c. Before the Standby mode is entered some initialization has to be done.

- Configuration of Standby Mode
- Configuration of Wakeup Timer
  - Source (Timer, ext. event)
  - Timer Interval
  - Wakeup Behavior
  - Timer Divider
- Configuration of ESRx

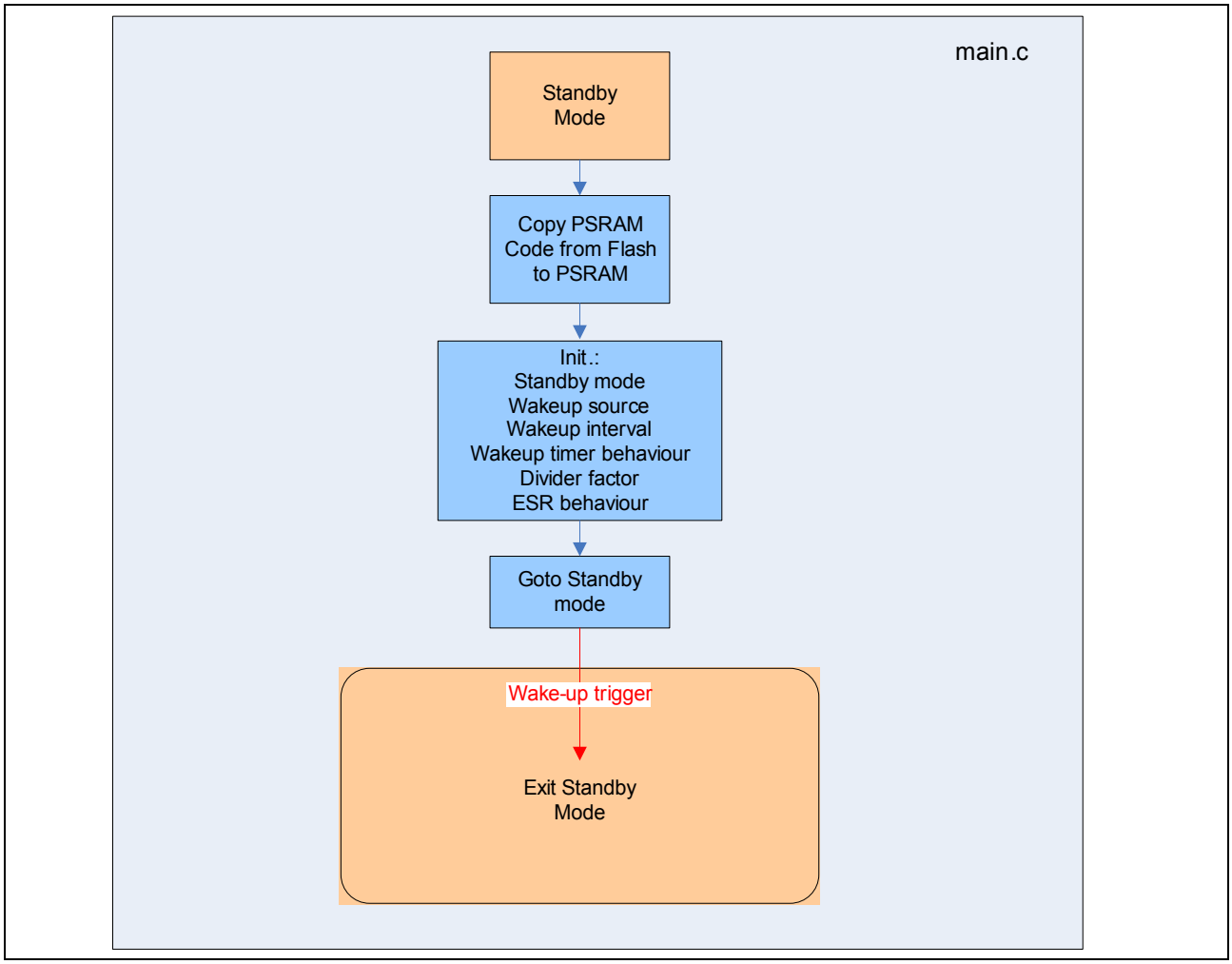

Figure 9 Standby Normal Flow in Function main

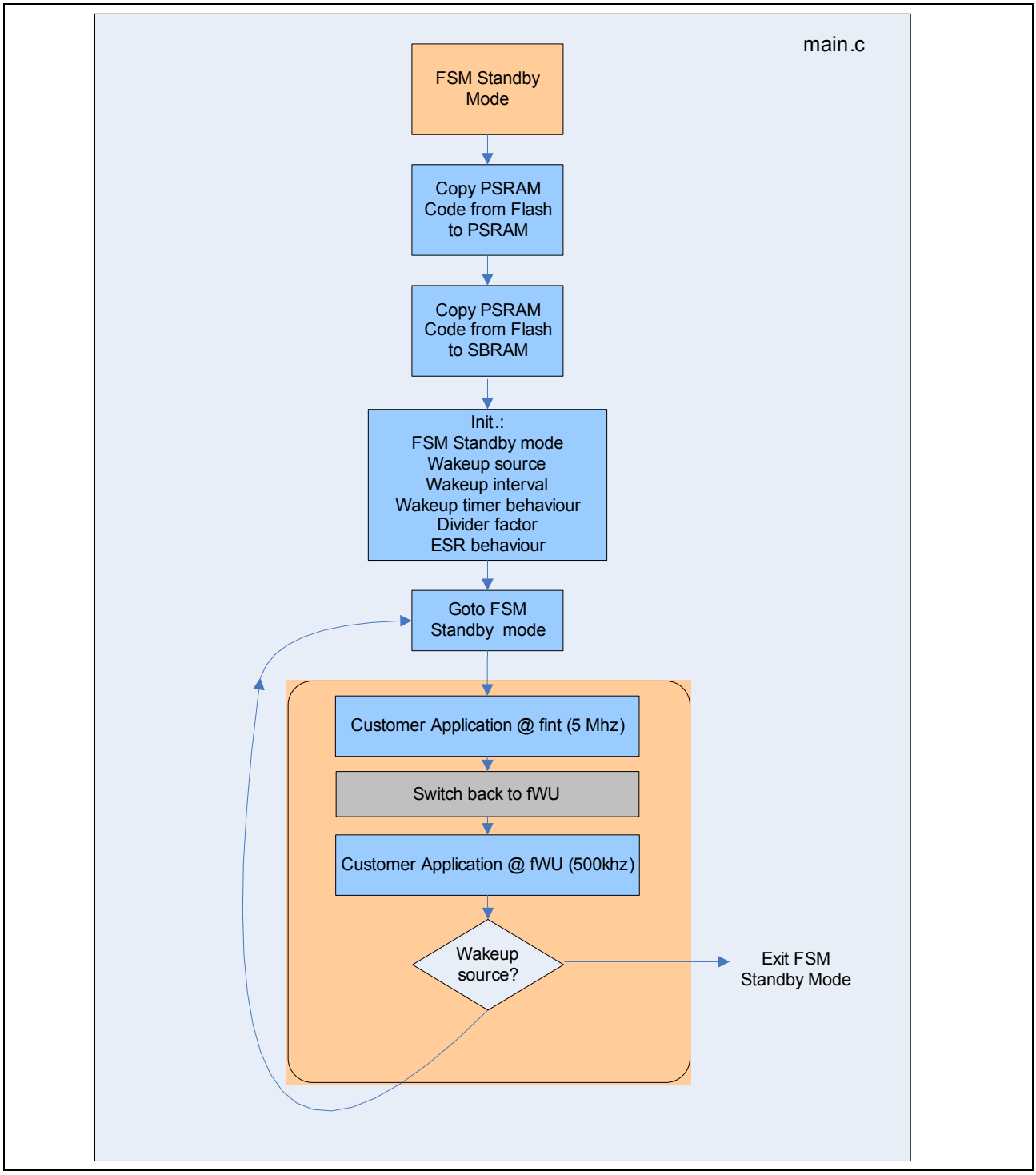

Figure 10 Standby FSM Flow in Function main

Depending on their requirements, the customer can execute the application with wakeup timer clock or internal clock.

2.3.1 Normal Standby Mode used

Table 14 Normal Standby Mode used

Name:	SCU_STANDBY_NORMAL_USED
Description:	Normal Standby Mode can be used
Type:	(int)
Default Value:	None
Dependency:	None
Range:	1 = mode can be used or 0 = mode cannot be used

Note: See Normal Stop-Over Mode used.

2.3.2 FSM Standby Mode used

Table 15 FSM Standby Mode used

Name:	SCU_STANDBY_FSM_USED
Description:	Standby Mode with Fast Startup Mode can be used
Type:	(int)
Default Value:	None
Dependency:	For newer devices only
Range:	1 = mode can be used or 0 = mode cannot be used

Note: See Normal Stop-Over Mode used.

2.3.3 SWD in Standby Mode

Table 16 SWD in Standby Mode

Name:	SCU_SWD_IN_STANDBY
Description:	Supply Watchdog state in normal and FSM Standby Mode
Type:	(int)
Default Value:	None
Dependency:	SCU_STANDBY_NORMAL_USED, SCU_STANDBY_FSM_USED
Range:	1 = SWD on in Standby Mode or 0 = SWD off in Standby Mode

Note: Disabling the supply watchdog in Standby Mode saves current.

Note: Option is only evaluated when a Standby Mode is used.

2.3.4 ULPEVR in Standby Mode

Table 17 ULPEVR in Standby Mode

Name:	SCU_ULPEVR_IN_STANDBY
Description:	Ultra low power EVR (ULPEVR) state in normal and FSM standby mode
Type:	(int)
Default Value:	None
Dependency:	SCU_CONTROLLER, SCU_STANDBY_NORMAL_USED, SCU_STANDBY_FSM_USED
Range:	1 = ULPEVR on in Standby Mode or 0 = ULPEVR off in Standby Mode

Note: Enabling the ultra low power EVR in Standby Mode saves current.

Note: Option is only evaluated if Value Line controller or higher is selected and if any standby mode is used.

2.4 Resources and Timing

Depending on the configuration, the SCU driver needs several resources.

Table 18 shows the memory occupied by the driver.

Table 18 Needed Resources

Memory	Standby	FSM	Stopover
Flash	4 KByte	3.2 KByte	3,7 KByte
PSRAM	0.9 KByte	0.6 KByte	1.0 KByte
SBRAM	-	150 Byte	-

Note: The occupied resource does not take into acoount the user specific application code.

As mentioned previously, parts of the SCU driver need delay algorithms to be sure that all configurations become stable. Table 19 and 20 roughly show the timing condition for a wakeup timer frequency of 500 KHz.

Table 19 Response Time

Normal Mode	Preparation Power Down Mode
~ 1.5ms @ 10MHz	~ 600µs @ 500KHz

Table 20 Response Time

Powerdown Modes	Standby @ 500 KHz	FSM @ 500 KHz	Stopover @ 500 KHz
Wakeup	~ 65µs + Boot time	~ 500µs	~ 160µs
Sleep	-	~ 65µs	~ 65µs

3 Hints for Tool Chain Usage

3.1 Supported Tool Chains

The following tool chains are supported:

- Tasking C166 VX
- Tasking C166 Classic
- Keil C166 (limited)

3.2 Using the Tasking VX Tool Chain

3.2.1 Importing the SCU Driver

There are two options for importing the SCU Driver to your system:

- Import the whole project:
  Create a new directory "ScuV<x>" in your work-space and copy all files into this directory.
  Via the Tasking VX GUI, import the whole project by using "File -> Import -> General -> Existing Projects into workspace".
  The source files and the tool chain settings will be copied.
- Import single files (for experienced users):
  Copy the files **Scu.c**, **Scu.h** and **Scu_Cfg.h** into an existing project.
  Copy/paste the required parts of Test.c and ScuV<x>.lsl into your existing source/linker script file.
  Existing startup files and tool chain settings will be kept.

3.2.2 Memory related Settings and Operations

The special memory related settings are contained in the linker script file.

Note that the settings, particularly the addresses, must be adapted to the needs of the application.

To mark the start and end points of a PSRAM or PSRAM-SBRAM code range, specific #pragmas have to be used; see file Test.c.

The copy actions, from Flash to PSRAM, and from Flash to SBRAM, are started during runtime via user software; see file Test.c.

3.2.3 Inappropriate Options for Tasking C166 VX Compiler

The following Compiler optimization techniques are not recommended for SCU code:

- Code Compaction:
  - The Compiler may replace an inline function that shall be executed in PSRAM by a real function which is located in Flash.
- Automatic Function Inlining:
  - The Compiler may replace a real function that shall be executed in PSRAM by inline code which is located in Flash.

Care must be taken that the above options are disabled for all functions that call the SCU Driver or are called by it, such as in Test.c. These options are therefore disabled in the file Scu.h via #pragmas.

3.3 Usage of Tasking Classic Tool Chain

It is strongly recommended to use the pre-configured project option file (*.opt). Advanced users can manually configure the settings.

3.3.1 Memory related Settings and Operations

It is assumed that the typical memory related settings (for on-chip Flash, or on-chip RAM for example) are made according to the chip used. With the EDE, this can be handled automatically.

When using EDE or *.ilo-File, additional memory related settings have to be made if power-saving features are used.

Table 21 Example for additional Memory related Settings

Setting (Example)	Explanation
MEMORY(ROM(0E00000h TO 0E009FFh))	PSRAM code range for total SCU code to be executed in PSRAM
RESERVE MEMORY(0C02000h TO 0C029FFh)	Flash range for total SCU code to be executed in PSRAM
CLASSES(SCU_CODE_PSRAM_SBRAM1, SCU_CODE_PSRAM_SBRAM2(0E00000h TO 0E001FFh UNIQUE))	SCU code to be executed in PSRAM, loaded from SBRAM (for standby mode only)
CLASSES(SCU_CODE_PSRAM(0E00200h TO 0E009FFh UNIQUE))	SCU code to be loaded/executed in PSRAM
ORDER SECTIONS(* 'SCU_CODE_PSRAM_SBRAM1', * 'SCU_CODE_PSRAM_SBRAM2')	Determine locate order (SCU_CODE_PSRAM_SBRAM1 starts at 0E00000h)

An Intel-Hex file is required as additional output format.

Using the Intel-Hex converter options, the range 0E00000 to 0E00A00 in the example above, must be remapped to the reserved Flash range with a start address of 0C02000.

Note: The Intel-Hex converter uses a different notation for addresses from the one used elsewhere.

The Intel-Hex file is used for programming the Flash. For debugging purposes, the *.out file has to be loaded into the debugger (without re-programming).

Note: The settings, particularly the addresses, must be adapted to the needs of the application.

To mark the start and end of a PSRAM or PSRAM_SBRAM code range, specific #pragmas have to be used. Refer to the file Test.c.

The copy actions, from Flash to PSRAM and from Flash to SBRAM, are started during runtime via user software; see file Test.c. The start and end addresses have to be defined there, and must be consistent with the tool chain settings.

Care should be taken when the smart linking option is used (this option is disabled by default). Code with class SCU_CODE_PSRAM_SBRAM1 (which is never called explicitly) must not be deleted by this option.

3.3.2 Other Settings

It is assumed that the typical memory related settings (for on-chip Flash, or on-chip RAM for example) are already made.

- Select appropriate memory model; For example, small/paged with page numbers 300h, 301h, 2h

- Include default register definition header file before source

- Set correct base address for interrupt vector table

4 Application Example

4.1 Functional Description

This example demonstrates a typical application use case:; An LED is switched on for 50 ms in a time interval of one second.

To reduce the overall power consumption, the microcontroller is driven in the stop-over mode.

The wakeup timer is used for the wake up trigger.

The LED on time is generated with a software counter.

- Microcontroller: XC2287M
- Normal Mode: Fcpu = 80 MHz
- Stop Over Mode: Fcpu = 500 KHz

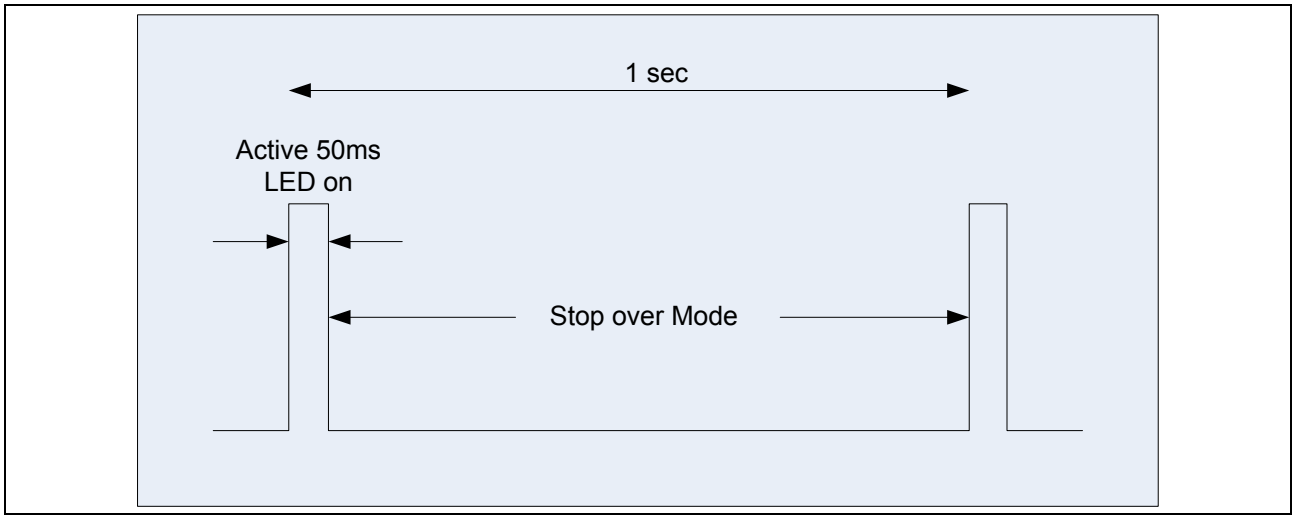

Figure 11 Application Scenario in Stop-Over Mode

The following configuration settings are made in **SCU_Cfg.h** (see the Figure that follows)>

```
/* 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 1
        #define SCU_STOPOVER_CRYSTAL_ON_USED 0
        #define SCU_STANDBY_NORMAL_USED0
        #define SCU_STANDBY_FSM_USED 0
        /* 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
        /* 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 12 Configuration in Scu_Cfg.h

In **stop_over_mode.c,** the following application code is required:

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

  /* get wake-up source */
  WakeupSrc = Scu_GetWakeupSrc();

  /* clear wake-up source */
  Scu_ClearWakeupSrc(WakeupSrc);

  if(WakeupSrc & SCU_WAKEUP_TIMER)
  {
         /*         Application code starts       */
         Test_ClearLedPin(0);
         /* delay =            (((delay[us] * fWU[kHz])          / 1000)/10 */
         Test_Delay((unsigned int)(((50000UL * (SCU_F_WU/1000U)) / 1000U)/10U));
         Test_SetLedPin(0);
         /*         Application code ends       */


    /* continue stop-over mode */
    return 1U;
  }
  else
  {
    /* ESR wake-up: */


    /* terminate stop-over mode */
    return 0U;
  }

} /* end of function Scu_HandleStopover_Ps */
#endif
/* (SCU_STOPOVER_CRYSTAL_ON_USED || SCU_STOPOVER_NORMAL_USED) */

/* end of code to be executed from PSRAM */
#if(SCU_COMPILER == SCU_COMPILER_TASKING_VX)
#  pragma section default
#endif

#endif /* (SCU_STOPOVER_CRYSTAL_ON_USED || SCU_STOPOVER_NORMAL_USED) */
```

Figure 13 Function Scu_HandleStopover_Ps (parts of stop_over_mode.c)

5 Limitations and Assumptions

5.1 Limitations

5.1.1 No Full Configuration Check

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

5.1.2 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

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 section 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.

Note: The SCU Driver itself will not cause any of the problems described above.

5.1.3 Known Clock Issues

No special Handling of external Clock at Crystal Input

If an external clock is connected to the XTAL1 input, the waiting time for a high precision oscillator might be removed.

6 Conclusion

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

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

Active Modes:

- Normal Mode
- Internal Clock Mode
- Direct Drive Mode

Power Down modes:

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

The Driver has been optimized in terms of:

- Robust design
- Error management

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.

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

Published by Infineon Technologies AG