TriCore™ AURIX™ Family

32-bit Micro-Controllers

Register C Header Files

General Usage Hints with Examples

User Manual

V1.5 2016-07

Microcontrollers

Edition 2016-07

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2016 Infineon Technologies AG
All Rights Reserved.

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.

Date	Version	Change Description
23.07.2014	1.0	Initial Creation
24.08.2015	1.1	Added section Do’s and Don’ts
21.09.2015	1.2	Added sections for AoU, added example for bitfield mask access
17.06.2016	1.3	Added sections for AoU, added for HW redesign for CANFD.
27.06.2016	1.4	Updated review comment #REV_010278.
15.07.2016	1.5	Updated AoU section for #REV_010340.

Trademarks

Infineon® is a registered trademark of Infineon Technologies Ltd.

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) to:

ctdd@infineon.com

Table of Contents

1 Introduction

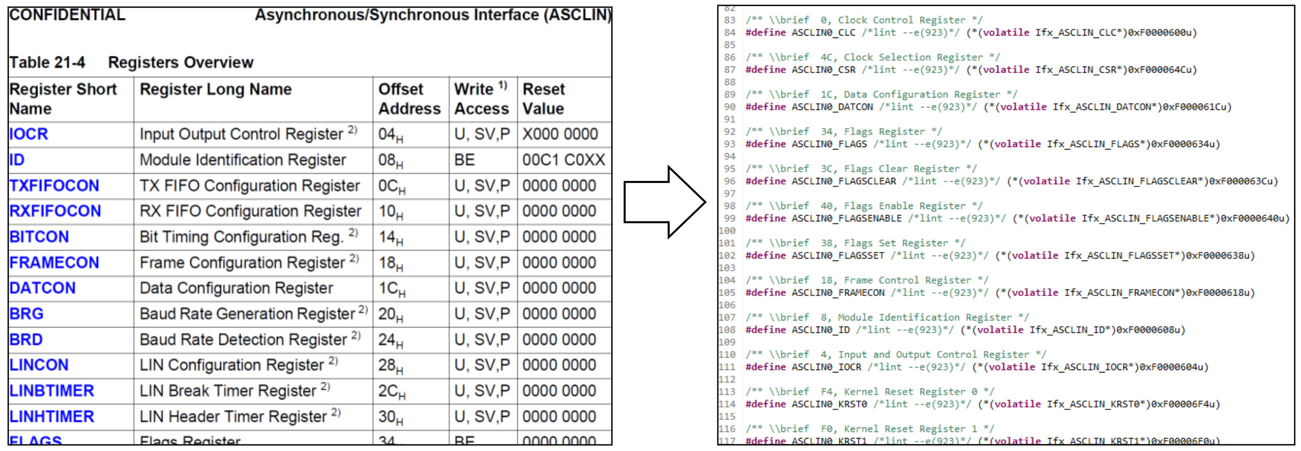
The product, “Register C Header Files” is the bridge to connect software domain with microcontroller hardware domain.

Software accessibility to microcontroller hardware is, always, only through the registers. These are commonly called as SFR (Special Function Registers). These registers in a way carry the commands, configuration or status to-and-from the microcontroller hardware. Each of the registers contains important attributes called as “bit-fields”. Each bit-field of a register represents specific functionality. User-manual of microcontroller hardware product, states the accessible registers along with their bit-field and states the functionality of each.

As the microcontroller hardware is very sensitive to the information these bit-fields carry, the user commands or configuration. Software functionalities are also sensitive to these bit-fields which carry the status information. Because of this sensitivity, the registers with their bit-fields are crucial for both the domains, i.e. software and hardware, to function together to fulfil the intended functionality.

Register C header files are C language representation of microcontroller registers because most of the software functionality are constructed with the C language. (On the other hand the header files for assembly language are Register ASM headers)

This product targets to represent only C language and not for assembly language usage.



CONFIDENTIAL Asynchronous/Synchronous Interface (ASCLIN)

Table 21-4 Registers Overview

Register Short Name	Register Long Name	Offset Address	Write [1] Access	Reset Value
IOCR	Input Output Control Register [2]	04_H	U, SV,P	X000 0000
ID	Module Identification Register	08_H	BE	00C1 C0XX
TXFIFOCON	TX FIFO Configuration Register	$0C_H$	U, SV,P	0000 0000
RXFIFOCON	RX FIFO Configuration Register	10_H	U, SV,P	0000 0000
BITCON	Bit Timing Configuration Reg. [2]	14_H	U, SV,P	0000 0000
FRAMECON	Frame Configuration Register [2]	18_H	U, SV,P	0000 0000
DATCON	Data Configuration Register	$1C_H$	U, SV,P	0000 0000
BRG	Baud Rate Generation Register [2]	20_H	U, SV,P	0000 0000
BRD	Baud Rate Detection Register [2]	24_H	U, SV,P	0000 0000
LINCON	LIN Configuration Register [2]	28_H	U, SV,P	0000 0000
LINBTIMER	LIN Break Timer Register [2]	$2C_H$	U, SV,P	0000 0000
LINHTIMER	LIN Header Timer Register [2]	30_H	U, SV,P	0000 0000
FLAGS	Flags Register	34	BE	0000 0000

User manual representation of registers’ memory map

```
/** \\brief  0, Clock Control Register */
#define ASCLIN0_CLC /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_CLC*)0xF0000600u)

/** \\brief  4C, Clock Selection Register */
#define ASCLIN0_CSR /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_CSR*)0xF000064Cu)

/** \\brief  1C, Data Configuration Register */
#define ASCLIN0_DATCON /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_DATCON*)0xF000061Cu)

/** \\brief  34, Flags Register */
#define ASCLIN0_FLAGS /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_FLAGS*)0xF0000634u)

/** \\brief  3C, Flags Clear Register */
#define ASCLIN0_FLAGSCLEAR /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_FLAGSCLEAR*)0xF000063Cu)

/** \\brief  40, Flags Enable Register */
#define ASCLIN0_FLAGSENABLE /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_FLAGSENABLE*)0xF0000640u)

/** \\brief  38, Flags Set Register */
#define ASCLIN0_FLAGSSET /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_FLAGSSET*)0xF0000638u)

/** \\brief  18, Frame Control Register */
#define ASCLIN0_FRAMECON /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_FRAMECON*)0xF0000618u)

/** \\brief  8, Module Identification Register */
#define ASCLIN0_ID /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_ID*)0xF0000608u)

/** \\brief  4, Input and Output Control Register */
#define ASCLIN0_IOCR /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_IOCR*)0xF0000604u)

/** \\brief  F4, Kernel Reset Register 0 */
#define ASCLIN0_KRST0 /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_KRST0*)0xF00006F4u)

/** \\brief  F0, Kernel Reset Register 1 */
#define ASCLIN0_KRST1 /*lint --e(923)*/ (*(volatile Ifx_ASCLIN_KRST1*)0xF00006F0u)
```

C Header file representation of registers’ memory map

Figure 1 Register memory map representations

1.1 Scope of the document

This user manual provides the overview, how to use Register C Headers in general with any of the targeted microcontroller product. However there are some examples which may carry specific register name from specific product. These are to be treated ONLY as examples and may or may not be correct to their actual HW function in a product.

2 Acronyms and abbreviations

Table 1 Acronyms and abbreviations

Acronym/ Abbreviation	Description
µC	Microcontroller
API	Application Programming Interface
HW	Hardware
IFX	Infineon Technologies
MCU	Micro Controller Unit
SW	Software
SFR	Special Function Registers

3 'Including' Register C Headers

This section provide the overview about the package structure, file naming conventions and the including the register headers in different used cases

3.1 Package Structure

A separate Register C Header Files package is delivered for each µC derivative. User must choose correct package for the target µC.

The release contains individual packages for each µC. These are an installer packages with the name same as their version tag

REG_TC27XB_UM_V1.4.R3.exe

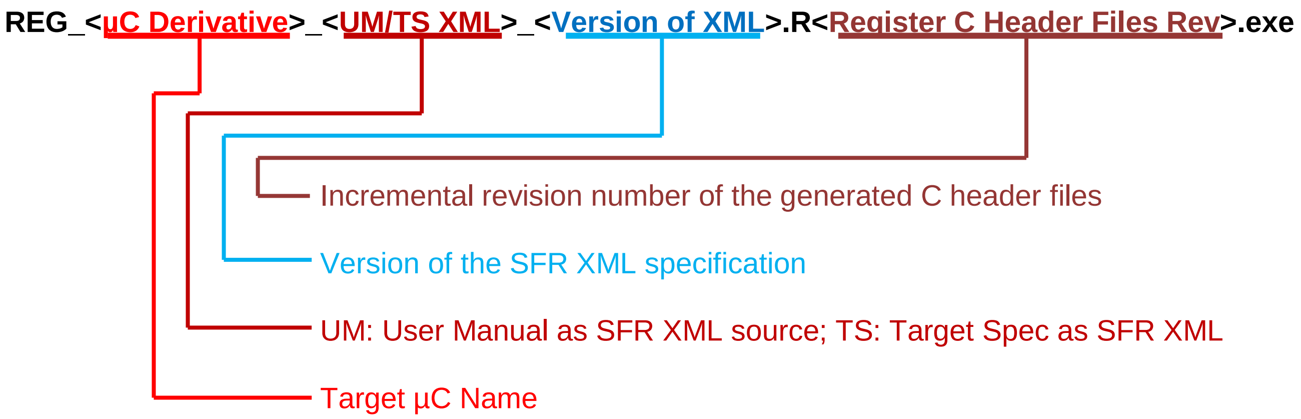

REG_<µC Derivative>_<UM/TS XML>_<Version of XML>.R<Register C Header Files Rev>.exe
Incremental revision number of the generated C header files
Version of the SFR XML specification
UM: User Manual as SFR XML source; TS: Target Spec as SFR XML
Target µC Name

Figure 2 Register C Header release package name tag

For example:

Package REG_TC27XB_UM_V1.4.R3.exe :

- Contains registers for TC27xB derivative
- The registers are generated from UM SFR XML file version 1.4 (which corresponds to hardware user manual version number 1.4)
- This is third incremental release for the generated registers from the same SF XML

Package once installed contains following folders

1) **_Reg**: This folder contains actual C header files, which are to be included in to the applications. This folder also contains the file _Package.xml to provide the logistic information about the files.
2) **SupportDocuments**: This folder contains the documents to support the user during development and project quality audits. This user manual is also part of the SupportDocuments.

3.2 Register C Header Files' naming

For each peripheral with in the µC, Register C representation is realized with three different files.

1) REGDEF files: Definition of Register/Peripheral Module Types: Representation of hardware register memory map or bit-field map for any peripheral are done through structure and union type definitions

   Naming convention: Ifx<peripheral name>_regdef.h

Where "peripheral name" represents the peripheral IP name of µC

Examples: IfxCpu_regdef.h, IfxCan_regdef.h, IfxEth_regdef.h

For peripheral IPs such as MultiCAN, the generic name CAN is used

2) REG files: Definition of register memory map: Register memory map representation of any peripheral is done through #define macros assigning the register names to their memory address.

   Naming convention: Ifx<peripheral name>_reg.h

   Where "peripheral name" represents the peripheral IP name of µC

   Examples: IfxCpu_reg.h, IfxCan_reg.h, IfxEth_reg.h

3) BF files: Definitions of register bit-fields to use the masked way of register access are done in BF files. These files contain the register bit-field mask, bit-field length and bit-field position.

   Naming convention: Ifx<peripheral name>_bf.h

   Where "peripheral name" represents the peripheral IP name of µC

   Examples: IfxCpu_bf.h, IfxCan_bf.h, IfxEth_bf.h

4) SUPERSET REG file: Include all the individual peripherals registers in one file. This is useful when an application use the registers from many different modules.

   File Name: Ifx_reg.h

5) BASICTYPES: Define the basic data types for the Registers' bit-fields. Some of the registers need the special keyword to tell the compiler that, such registers' bit-fields can only be accessed as 32 bits or 16 bits. Compiler will not optimize access to such register bit-fields.

   File Name: Ifx_TypesReg.h

3.3 Include Structure with in the Register C Header files

Register headers files have fixed include structure hierarchy in such a way that user has flexibility and ease of use across different used cases. Figure 3 below show this hierarchy.

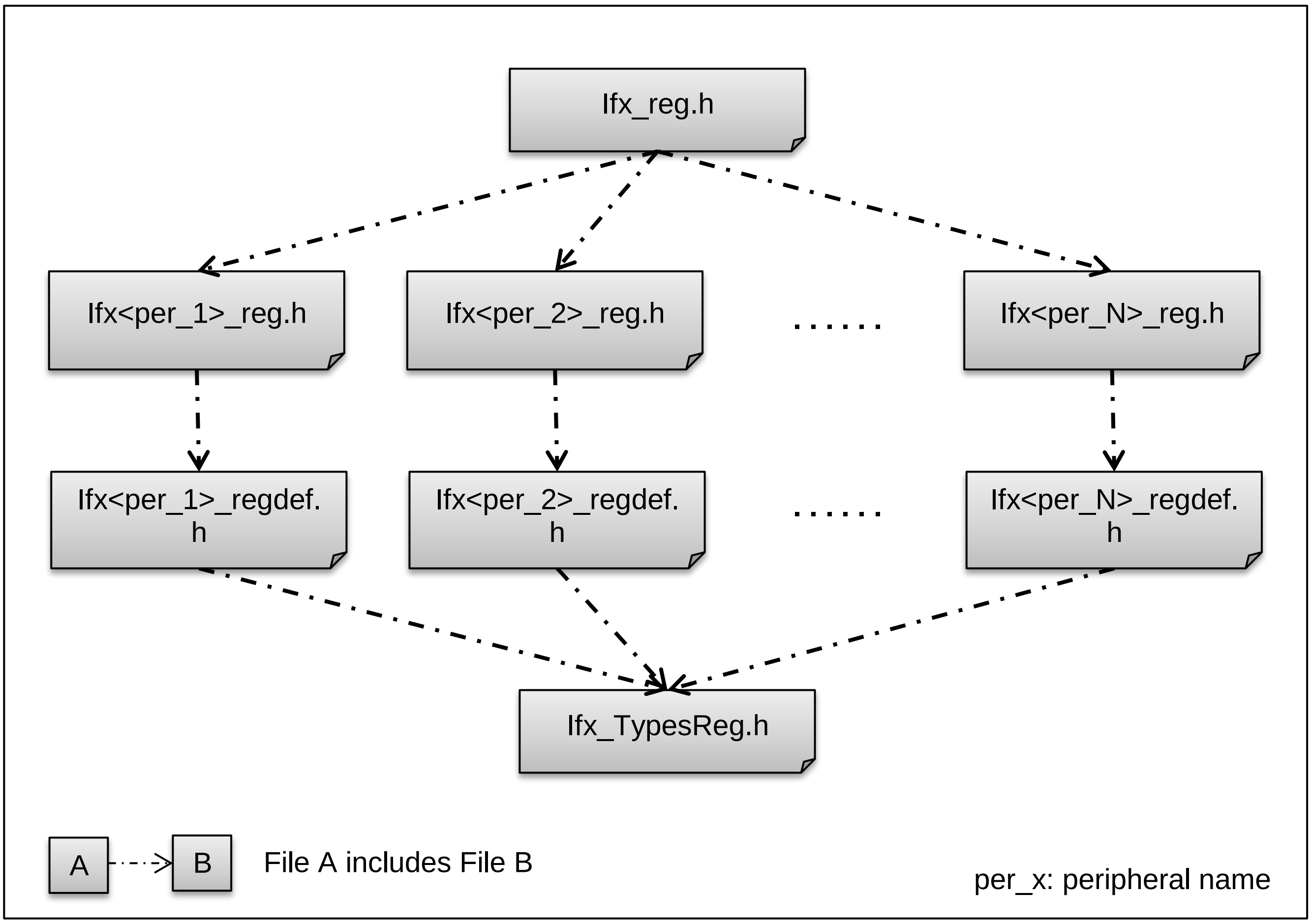


Figure 3 Include Structure with in Register C Header files

Files, Ifx<peripheral name>_bf.h, are individual files which are independently included based on the need in the application

3.4 Hints on Including the Register Header files

To access Registers, only Ifx_reg.h and Ifx<peripheral name>_bf.h are required to be included by the user. Other files are indirectly included as shown in Figure 3 Include Structure with in Register C Header files, above.

This section provide the details about including the Register C Header Files based on the used cases

3.4.1 Usage with peripheral driver files

When a driver is developed for a specific peripheral HW module, registers from only few the peripheral are accessed. It is recommended to include only the files for to the needed peripheral modules. For example for ADC driver for VADC peripheral include only include IfxVadc_reg.h and additionally IfxScu_reg.h.

The above is recommended for optimizing the compile time of individual driver files

3.4.2 Access SFRs from more than one peripheral

When an application, a test functionality/ start up functionality needs to access multiple peripherals, then it is easy to include Ifx_reg.h file.

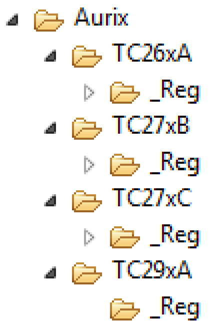
Compile time is affected only for the individual application file

3.4.3 Handling different derivatives

Register C Header files are available as specific package for each microcontroller derivative. When a generic driver targeted for multiple µC derivatives, it is important to take care that while build process, correct set of files for the targeted µC are included.

To handle this, following is one of the recommendations:

1) Place the _Reg folders in peripheral specific folders as below example



- Aurix
  - TC26xA
    - _Reg
  - TC27xB
    - _Reg
  - TC27xC
    - _Reg
  - TC29xA
    - _Reg

2) Include the register header files without their paths. For example:

Code Listing 1

```
#include "IfxVadc_reg.h"
#include "IfxScu_reg.h"
```

3) For target specific build, configure the include path to ../Aurix/< µC target>_Reg. For example for the target TC27xC controller compiler option shall be: -I./<path>/Aurix/TC27xC/_Reg.

4 Accessing individual registers

Register headers provide an interface to access the HW. Most of the cases require to access the register as volatile access (volatile is compiler directive for defining a data to tell compiler not to optimize the code but for each read/write to SFR, directly access the memory)

Such an access of registers for a peripheral draws following scenarios

4.1 Access the register targeting an individual bit-field

For a driver function if only one of the bit-fields need to be write accessed, this can be done as directly assigning the value to the bit-field.

Following is the syntax:

<SFR Name>.B.<Bit-field Name>= <value to be assigned>

For example:

The driver needs writing to EDIS bit of STM0_CLC

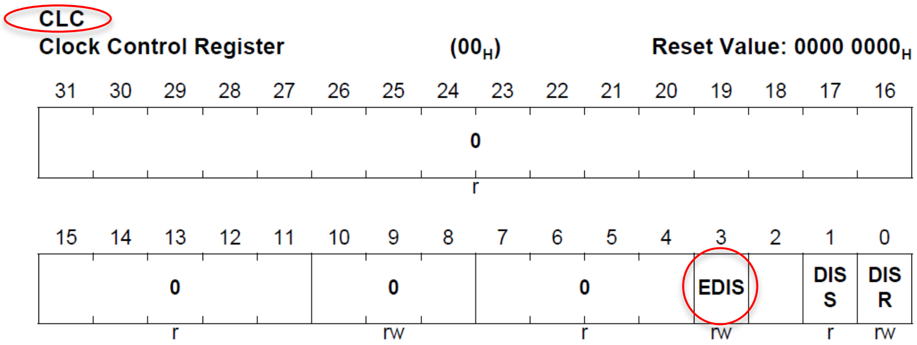


Figure 4 User Manual representation of Register STM0_CLC

Below is the code snippet:

Code Listing 2

```
#include “IfxStm_reg.h”

/*Example function*/
void myExample_accessReg_01(void)
{
    STM0_CLC.B.EDIS= 1;
}
```

4.2 Access the register targeting multiple bit-fields

For a driver function if multiple bit-fields need to be write accessed, this can be done in many ways as below:

1) Assigning the values to the all bit-fields individually
2) Use the bit-field mask and position to make a read modify write operation

Both the above approaches are having disadvantages as approach 1) is not optimized to access the SFR memory each time and approach 2) is not producing a readable code.

To solve both problems following is the recommendation:

Make a function local variable of targeted register type. As type-define of target register is not with volatile keyword all the bit-fields access will be with local register access.

Please also note that, the type define without volatile keyword has this specific used case in mind. To access register as volatile access one must use the register name directly.

Assign values to individual bit-fields in the same way as writing to register bit-field, but with local variable.

After all the bit-fields are written, assign to the actual register accessing it as unsigned Integer.

For example:

The driver needs to write to SCU_CCUCON1 register bit fields.

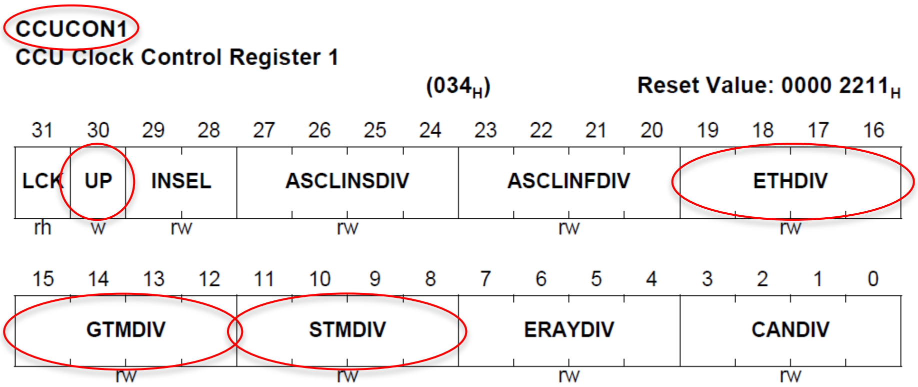


Figure 5 User Manual representation of Register SCU_CCUCON1

Below is the code snippet for above access:

Code Listing 3

```
#include “IfxScu_reg.h”

/*Example function*/
void myExample_accessReg_02(void)
{
  Ifx_SCU_CCUCON1 myLocalReg;       /*nonvolatile definition*/
  myLocalReg.U = SCU_CCUCON1.U;     /*Read the register value locally*/

  /*Write to local variable*/
  myLocalReg.B.ETHDIV= 1;
  myLocalReg.B.GTMDIV= 1;
  myLocalReg.B.STMDIV= 1;
  myLocalReg.B.UP= 1;               /*Set the update bit*/
  SCU_CCUCON1.U= myLocalReg.U;      /*volatile access*/
}
```

4.3 Access the registers with its bit mask values

Registers could be accessed also with the bit mask values. This approach could be used for following used cases

1) Used case required efficient access of the registers.

2) The target register has some reserved bitfields which are need to be written with non-zero values.
3) Used case requires accessing the registers with atomic load-modify-store instructions.

Please note that the file: Ifx<peripheral name>_bf.h shall be individually and explicitly included. And they are not implicitly included by Ifx<peripheral name>_reg.h or Ifx_reg.h.

For the used case 3 above, following code provide an example. This example access same register with same values, as in 4.2 above.

Code Listing 4

```
#include "IfxScu_bf.h"

/*Example function*/
void myExample_accessReg_03(void)
{
  uint32 myMask=IFX_SCU_CCUCON1_ETHDIV_MSK << IFX_SCU_CCUCON1_ETHDIV_OFF |
                IFX_SCU_CCUCON1_GTMDIV_MSK << IFX_SCU_CCUCON1_GTMDIV_OFF |
                IFX_SCU_CCUCON1_STMDIV_MSK << IFX_SCU_CCUCON1_STMDIV_OFF |
                IFX_SCU_CCUCON1_UP_MSK << IFX_SCU_CCUCON1_UP_OFF;

  uint32 myVal= 1 << IFX_SCU_CCUCON1_ETHDIV_OFF |
                1 << IFX_SCU_CCUCON1_GTMDIV_OFF |
                1 << IFX_SCU_CCUCON1_STMDIV_OFF |
                1 << IFX_SCU_CCUCON1_UP_OFF;

  __ldmst((void *)(&SCU_CCUCON1), myMask, myVal);
}
```

5 Accessing registers through grouped structures

Linear memory map of the HW Registers can be translated to logical groups with the same hierarchy, as they appear in the HW. Such grouping is represented in C as, structures' instances and array of either (such) instances or array of individual registers themselves. This grouping eases the development of drivers with error free and efficient coding and results in optimized code.

This kind of usage is directly explained with the below example:

Example: Accessing multiple registers of a STM and the instance is configurable.

Here the module instance can be passed as a pointer of type of that module.

Code Listing 5

```
#include “IfxStm_reg.h”

/*Example function*/
void myExample_initStmReg_01(Ifx_STM *stm)
{
  stm->CMP[0].B.CMPVAL= 0xFFFF; /*Update compare value*/
  stm->ICR.B.CMP0EN= 1; /*Enable compare*/
}
```

For modules which have logical grouping for module sub modules and sub-sub modules etc., the same approach can be used.

As another example, the group can also be used to iterate through the array of logical instance such as CAN node.

6 Specific hints

Following are some of the hints when register header files are used for safety applications

1) Include only the needed register header files for the target peripheral
2) Don't rely only on the comments against the registers/ bit-fields to understand the behavior of the HW. The comments are currently not verifiable. Instead, refer to the user manual of the target HW.
3) Before accessing the register, user must check, if the register or bitfield is implemented in the used instance of target peripheral. The structures defined normally contain the superset of registers / bitfields. (for e.g. IOCR8 registers are not available for P10, however if user try to access it with MODULE_P10.IOCR8.U could be accessed in SW)
4) Register C Headers are tightly linked to the hardware user manual, based on which it is generated, as detailed in section 3.1 above. Any Errata corresponding to the hardware user manual may have impact on the register or bit field definitions. Users must take due care also while using such registers.

6.1 Dos and Don'ts:

Following are the list of do's and Don'ts.

Table 2 Do's and Don'ts

Dos	Don'ts
Include Ifx<per>_reg.h	Don't include Ifx<per>_regdef.h Why?: This is internal file, file name/ internal include e.t.c could change
Use Register Data Types only for local variables	Don't define your own registers using register typedefs. Why? Portability is affected as the memory map changes derivative to derivative. The register typedefs are not defined with volatile keyword explicitly.
Use local variables to write multiple bitfields (if it is allowed from targeted function)	Don't use volatile write unnecessarily Why? It is un-optimized
Use register groups with MODULE_<per> structure where it is necessary	Don't define your own groups Why? Portability is affected as the memory map changes derivative to derivative.
Be choosy to include only required header file	Don't include Ifx_reg.h file or don't include an unnecessary header file. Why? It affects the compilation time

7 Assumptions of Use (AoU)

Register C Header files are intended to be used in safety related applications. Register C Headers are developed based on the following assumptions of use. It is user’s responsibility to take care that these assumptions are handled with due care, while using them.

1) When a register bitfield is accessed implicitly or explicitly, user is aware of the fact that accessed bitfields are either
    a. Available at the hardware target, or
    b. If not available, they are written with right values as defined by the hardware target user manual.
2) Register C Header files don’t implement the HW errata and HW documentation errata. Any applicable errata shall be respected by the user while using the affected register.

For example: CAN Errata Number: MultiCAN_TC.H008 , the NISO & PED bitfield/s not defined in SFRs, to use this register correctly the user must define (in user files/driver files) the mask and offsets and use them with _ldmst instruction as shown in section 4.3.

Below is an example code snippet:

Code Listing 6

```
#include "IfxCan_bf.h"

#define <driver Prefix>_N0_BTR_NISO_LEN    (1U)
#define <driver Prefix>_N0_BTR_NISO_MSK    (1U)
#define <driver Prefix>_N0_BTR_NISO_OFF    (15U)

/* Example to set ISO CAN FD format */
void myExample_accessReg_04(void)
{
  uint32 myMask= <driver Prefix>_N0_BTR_NISO_MSK <<
                 <driver Prefix>_N0_BTR_NISO_OFF ;

  uint32 myVal= 1 << <driver Prefix>_N0_BTR_NISO_OFF;

  __ldmst((void *)(&CAN_N0_BTR), myMask, myVal);
}
```

8 PC Lint + MISRA

Register header files has few listed deviations, as in Table 3 below against the MISRA. For Register C headers, MISRA 2004 compliance is checked with PC Lint tool.

The warnings are analyzed for register header files and deviations are noted down in a compiled report. User must be aware of such deviations and check the justifications if they are OK from the projects’ needs. Such lint reports are made available through:

1) *<install folder>/SupportDocuments/PcLint_Messages.txt* (Raw file for messages)and
2) *<install folder>/SupportDocuments/PCLint_Report.xls* (compiled report with justifications)

Following are list of deviations made with the register header files:

Table 3

PC lint No	MISRA No	Generic Violation Message + Justification
46; 960	6.4	**Violation**
		Field type should be int
		Justification
		Such message is for following #define macro `Ifx_Strict_16Bit <bit field name>:<bit position>;` The bit-fields of Aurix Mc registers are 16 bit width where as Aurix controllers are 32 bit controllers By defining "int" in place of "Ifx_Strict_16Bit" will make the register offset different than that of HW. Such definitions are allowed with compilers, which support Aurix µC
537	.	**Violation**
		Repeated include file 'File'
		Justification
		Each of the Ifx<peripheral name>_reg.h includes the Ifx_TypesReg.h Ifx_TypesReg.h is multiple include protected and there is no harm to the user program due to this
621	1.4	**Violation**
		Identifier clash (Symbol 'Name' with Symbol 'Name' at String)
		Justification
		Compilers which support Aurix have no limitation of 31 characters
658		**Violation**
		Anonymous union assumed
		Justification
		Anonymous unions are defined for the multi-view registers in CAN, VADC and GTM peripherals As these multi-view registers are different registers but point to same address in the register memory map for the better readability of the code where these registers are used, this is defined this way Aurix compilers support such feature

PC lint No	MISRA No	Generic Violation Message + Justification

923	11.1;
	11.3;
		Violation

		Cast from Type to Type -- A cast is being made either from a pointer to a non-pointer or from a non-pointer to a pointer
		Justification
		Actually this is not listed in rule 11.1 of MISRA-C Guidelines Oct 2004 document. There is no function pointers involved here. However this statement is in 11.3 which allows the used case in the register C Headers Reference: MISRA-C Guidelines Oct 2004 document, page 53. “Casting between a pointer and an integer type should be avoided where possible, but may be unavoidable when addressing memory mapped registers or other hardware specific features.” Such a typecast is converting to an object pointer from an integer type. **This message is suppressed at the code line**
960	19.4	**Violation**
		Violates MISRA 2004 Required Rule 19.4, Disallowed definition for macro '
		Justification
		These messages are for definitions format: `#define <reg/module name> (*(<struct type>*)0x<address>u)` Such defines expand to constant. This is allowed in MISRA Reference: MISRA-C Guidelines Oct 2004 document, page 79: the rule statement states: "C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct."
960	18.4	**Violation**
		Violates MISRA 2004 Required Rule 18.4, declaration of union type or object of union type:
		Justification
		Such message is for following kind of defines typedef union { unsigned int U; signed int I; <reg bit-field struct type> B; } <union typedef>; Deviations for SFR unions allowed by MISRA: Reference: MISRA-C Guidelines Oct 2004 document, page 76: "The use of deviations is acceptable for (a) packing and unpacking of data" further --- "bit-order - how are bits numbered within bytes and how are bits allocated to bit fields"

PC lint No	MISRA No	Generic Violation Message + Justification
960	20.2	**Violation**
		Violates MISRA 2004 Required Rule 20.2, Re-use of C90 identifier pattern
		Justification
		Any of the Bit-field types are not standard C90 identifier pattern, they are specific to Infineon register types prefixed with "Ifx" Reference: MISRA-C Guidelines Oct 2004 document, page 84: the Rule 20.2 (required) statement states: "The names of standard library macros, objects and functions shall not be reused."
970	6.3	**Violation**
		Use of modifier or type 'Name' outside of a typedef
		Justification
		All these definitions are of one of the formats: unsigned int/short/char U; signed int/short/char I; These definitions are for register unions. The deviations are justified based on following considerations: 1) The register header files are only used with Aurix controllers and for a particular derivative microcontroller 2) The register header files cannot be portable for any unsupported compiler 3) The supported compilers for Aurix controllers interpret the basic types int, short and char as 32bit, 16bit and 8bit values respectively

www.infineon.com

Published by Infineon Technologies AG