Application Report

SPRAA85E–November 2005–Revised December 2017

Programming TMS320x28xx and TMS320x28xxx Peripherals in C/C++

Lori Heustess, Whitney Dewey

C2000

ABSTRACT

This application report explores hardware abstraction layer implementations to make programming of peripherals easy using C/C++ on TMS320x28xx and TMS320x28xxx devices. The methods of using bit field structure header files and the C2000™ Peripheral Driver Library are compared to each other and to the traditional #define macro approach. Topics of code efficiency and special case registers are also addressed.

Contents

List of Figures

List of Tables

Trademarks

C2000, Piccolo, Delfino, Code Composer Studio are trademarks of Texas Instruments.
All other trademarks are the property of their respective owners.

1 Introduction

The TMS320x28xx and TMS320x28xxx are members of the C2000 family of microcontrollers (MCUs). These devices are targeted for embedded control applications. To facilitate writing efficient and easy-to-maintain embedded C/C++ code on these devices, Texas Instruments provides hardware abstraction layer methods for accessing memory-mapped peripheral registers. These methods are the bit field and register-file structure approach, and the C2000 Peripheral Driver Library approach. This application report explains the implementation of these hardware abstraction layers and compares them to traditional #define macros. Topics of code efficiency and special case registers are also addressed.

The bit field and register-file structure hardware abstraction layer discussed in this application report has been implemented as a collection of C/C++ header files available for download in C2000Ware™ from Texas Instruments:

Support for all new microcontrollers is available in the device support section of C2000Ware. At this time, it supports and is the preferred approach for the following devices:

- **Piccolo™ Series Microcontrollers**
- **Delfino™ Series Microcontrollers**
- **F28M3x Series Microcontrollers (C28x Subsystem)**

Older C28x devices are not supported by C2000Ware and are instead supported in the following downloads:

- *C281x C/C++ Header Files and Peripheral Examples*
- *C280x, C2801x C/C++ Header Files and Peripheral Examples*
- *C2804x C/C++ Header Files and Peripheral Examples*

The C2000 Peripheral Driver Library (often referred to as “Driverlib”) is also available for download in C2000Ware. At this time, it supports the following devices:

- **F2807x**
- **F28004x**
- **F2837xS**
- **F2837xD**

Depending on your current needs, the software included in these downloads are learning tools or the basis for a development platform.

- **Learning Tool:**

  The C/C++ Header Files and Peripheral Examples include several example Code Composer Studio™ projects. These examples explain the steps required to initialize the device and utilize the on-chip peripherals. The examples can be copied and modified to quickly experiment with peripheral configurations.

- **Development Platform:**

  The header files can be incorporated into a project as a hardware abstraction layer for accessing the on-chip peripherals using C or C++ code. You can also pick and choose functions as needed and discard the rest. This application report does not provide a tutorial on C, C++, C28x assembly, or emulation tools. You should have a basic understanding of C code and the ability to load and run code using Code Composer Studio. While knowledge of C28x assembly is not required to understand the hardware abstraction layer, it is useful to understand the code optimization and read-modify-write sections. If you have assembly instruction-related questions, see the *TMS320C28x CPU and Instruction Set Reference Guide*.

Examples are based on the following software versions:

- *C281x C/C++ Header Files and Peripheral Examples V1.00*
- *C280x, C2801x C/C++ Header Files and Peripheral Examples V1.41*

- C2804x C/C++ Header Files and Peripheral Examples V1.00
- C2000Ware 1.00.02.00
- C2000 Compiler v16.9.3.LTS

The following abbreviations are used:

- C/C++ Header Files and Peripheral Examples refers to any of the header file or device support packages.
- Driverlib refers to the C2000 Peripheral Driver Library.
- TMS320x280x and 280x refer to all devices in the TMS320x280x and TMS320x2801x family. For example: TMS320F2801, TMS320F2806, TMS320F2808, TMS320F28015 and TMS320F28016.
- TMS320x2804x and 2804x refers all devices in the TMS320x2804x family. For example, the TMS320F28044.
- TMS320x281x and 281x refer to all devices in the TMS320x281x family. For example: TMS320F2810, TMS320F2811, and TMS320F2812, TMS320C2810, and so forth.
- C28x refers to the TMS320C28x CPU; this CPU is used on all of the above DSPs.

2 Traditional #define Approach

Developers have traditionally used #define macros to access registers in C or C++. To illustrate this approach, consider the SCI-A and SCI-B register files shown in Table 1.

Table 1. SCI-A, SCI-B Configuration and Control Registers

SCI-A Register Name [1]	Address [2]	Description
SCICCRA	0x7050	SCI-A Communications Control Register
SCICTL1A	0x7051	SCI-A Control Register 1
SCIHBAUDA	0x7052	SCI-A Baud Register, High Bits
SCILBAUDA	0x7053	SCI-A Baud Register, Low Bits
SCICTL2A	0x7054	SCI-A Control Register 2
SCIRXSTA	0x7055	SCI-A Receive Status Register
SCIRXEMUA	0x7056	SCI-A Receive Emulation Data Buffer Register
SCIRXBUFA	0x7057	SCI-A Receive Data Buffer Register
SCITXBUFA	0x7059	SCI-A Transmit Data Buffer Register
SCIFFTXA	0x705A	SCI-A FIFO Transmit Register
SCIFFRXA	0x705B	SCI-A FIFO Receive Register
SCIFFCTA	0x705C	SCI-A FIFO Control Register
SCIPRIA	0x705F	SCI-A Priority Control Register
SCI-B Register Name [3]	**Address**	**Description**
SCICCRB	0x7750	SCI-B Communications Control Register
SCICTL1B	0x7751	SCI-B Control Register 1
SCIHBAUDB	0x7752	SCI-B Baud Register, High Bits
SCILBAUDB	0x7753	SCI-B Baud Register, Low Bits
SCICTL2B	0x7754	SCI-B Control Register 2
SCIRXSTB	0x7755	SCI-B Receive Status Register
SCIRXEMUB	0x7756	SCI-B Receive Emulation Data Buffer Register
SCIRXBUFB	0x7757	SCI-B Receive Data Buffer Register
SCITXBUFB	0x7759	SCI-B Transmit Data Buffer Register
SCIFFTXB	0x775A	SCI-B FIFO Transmit Register
SCIFFRXB	0x775B	SCI-B FIFO Receive Register
SCIFFCTB	0x775C	SCI-B FIFO Control Register
SCIPRIB	0x775F	SCI-B Priority Control Register

(1) These registers are described in the*TMS320x281x Serial Communications Interface (SCI) Reference Guide* (SPRU051).

(2) Actual addresses may differ from device to device. See the device's Technical Reference Manual for details.

(3) These registers are reserved on devices without the SCI-B peripheral, and some devices may have more than two instances of the SCI peripheral. For more details, see the device-specific data manual.

A developer can implement #define macros for the SCI peripherals by adding definitions like those in Example 1 to an application header file. These macros provide an address label, or a pointer, to each register location. Even if a peripheral is an identical copy a macro is defined for every register. For example, every register in SCI-A and SCI-B is specified separately.

Example 1. Traditional #define Macros

```
/********************************************************************
* Traditional header file
********************************************************************/

#define Uint16 unsigned int
#define Uint32 unsigned long
                                             // Memory Map
                                             // Addr   Register
#define SCICCRA   (volatile Uint16 *)0x7050  // 0x7050 SCI-A Communications Control
#define SCICTL1A  (volatile Uint16 *)0x7051  // 0x7051 SCI-A Control Register 1
#define SCIHBAUDA (volatile Uint16 *)0x7052  // 0x7052 SCI-A Baud Register, High Bits
#define SCILBAUDA (volatile Uint16 *)0x7053  // 0x7053 SCI-A Baud Register, Low Bits
#define SCICTL2A  (volatile Uint16 *)0x7054  // 0x7054 SCI-A Control Register 2
#define SCIRXSTA  (volatile Uint16 *)0x7055  // 0x7055 SCI-A Receive Status
#define SCIRXEMUA (volatile Uint16 *)0x7056  // 0x7056 SCI-A Receive Emulation Data Buffer
#define SCIRXBUFA (volatile Uint16 *)0x7057  // 0x7057 SCI-A Receive Data Buffer
#define SCITXBUFA (volatile Uint16 *)0x7059  // 0x7059 SCI-A Transmit Data Buffer
#define SCIFFTXA  (volatile Uint16 *)0x705A  // 0x705A SCI-A FIFO Transmit
#define SCIFFRXA  (volatile Uint16 *)0x705B  // 0x705B SCI-A FIFO Receive
#define SCIFFCTA  (volatile Uint16 *)0x705C  // 0x705C SCI-A FIFO Control
#define SCIPRIA   (volatile Uint16 *)0x705F  // 0x705F SCI-A Priority Control
#define SCICCRB   (volatile Uint16 *)0x7750  // 0x7750 SCI-B Communications Control
#define SCICTL1B  (volatile Uint16 *)0x7751  // 0x7751 SCI-B Control Register 1
#define SCIHBAUDB (volatile Uint16 *)0x7752  // 0x7752 SCI-B Baud Register, High Bits
#define SCILBAUDB (volatile Uint16 *)0x7753  // 0x7753 SCI-B Baud Register, Low Bits
#define SCICTL2B  (volatile Uint16 *)0x7754  // 0x7754 SCI-B Control Register 2
#define SCIRXSTB  (volatile Uint16 *)0x7755  // 0x7755 SCI-B Receive Status
#define SCIRXEMUB (volatile Uint16 *)0x7756  // 0x7756 SCI-B Receive Emulation Data Buffer
#define SCIRXBUFB (volatile Uint16 *)0x7757  // 0x7757 SCI-B Receive Data Buffer
#define SCITXBUFB (volatile Uint16 *)0x7759  // 0x7759 SCI-B Transmit Data Buffer
#define SCIFFTXB  (volatile Uint16 *)0x775A  // 0x775A SCI-B FIFO Transmit
#define SCIFFRXB  (volatile Uint16 *)0x775B  // 0x775B SCI-B FIFO Receive
#define SCIFFCTB  (volatile Uint16 *)0x775C  // 0x775C SCI-B FIFO Control
#define SCIPRIB   (volatile Uint16 *)0x775F  // 0x775F SCI-B Priority Control
```

Each macro definition can then be used as a pointer to the register's location as shown in Example 2.

Example 2. Accessing Registers Using #define Macros

```
/********************************************************************
* Source file using #define macros
********************************************************************/
...
   *SCICTL1A  = 0x0003;        //write entire register
   *SCICTL1B |= 0x0001;     //enable RX
...
```

Some advantages of traditional #define macros are:

- Macros are simple, fast, and easy to type.
- Variable names exactly match register names; variable names are easy to remember.

Disadvantages to traditional #define macros include the following:

- Bit fields are not easily accessible; you must generate masks to manipulate individual bits.
- You cannot easily display bit fields within the Code Composer Studio watch window.
- Macros do not take advantage of Code Composer Studio's auto-completion feature.
- Macros do not benefit from duplicate peripheral reuse.

3 Bit Field and Register-File Structure Approach

Instead of accessing registers using #define macros, it is more flexible and efficient to use a bit field and register-file structure approach.

- **Register-File Structures:**

  A register file is the collection of registers belonging to a peripheral. These registers are grouped together in C/C++ as members of a structure; this is called a register-file structure. Each register-file structure is mapped in memory directly over the peripheral registers at compile time. This mapping allows the compiler to efficiently access the registers using the CPU's data page pointer (DP).
- **Bit Field Definitions:**

  Bit fields can be used to assign a name and width to each functional field within a register. Registers defined in terms of bit fields allow the compiler to manipulate single elements within a register. For example, a flag can be read by referencing the bit field name corresponding to that flag.

The remainder of this section describes a register-file structure with bit-field implementation for the SCI peripherals. This process consists of the following steps:

1. Create a simple SCI register-file structure variable type; this implementation does not include bit fields.
2. Create a variable of this new type for each of the SCI instances.
3. Map the register-file structure variables to the first address of the registers using the linker.
4. Add bit-field definitions for select SCI registers.
5. Add union definitions to provide access to either bit fields or the entire register.
6. Rewrite the register-file structure type to include the bit-field and union definitions.

In the C/C++ Header Files and Peripheral Examples, the register-file structures and bit fields have been implemented for all peripherals on the C28x cores of the TMS320x28xx and TMS320x28xxx devices.

3.1 Defining A Register-File Structure

Example 1 showed a hardware abstraction implementation using #define macros. In this section, the implementation is changed to a simple register file structure. Table 2 lists the registers that belong to the SCI peripheral. This register file is identical for each instance of the SCI, i.e., SCI-A and SCI-B.

Table 2. SCI-A and SCI-B Common Register File

Name	Size	Address Offset	Description
SCICCR	16 bits	0	SCI Communications Control Register
SCICTL1	16 bits	1	SCI Control Register 1
SCIHBAUD	16 bits	2	SCI Baud Register, High Bits
SCILBAUD	16 bits	3	SCI Baud Register, Low Bits
SCICTL2	16 bits	4	SCI Control Register 2
SCIRXST	16 bits	5	SCI Receive Status Register
SCIRXEMU	16 bits	6	SCI Receive Emulation Data Buffer Register
SCIRXBUF	16 bits	7	SCI Receive Data Buffer Register
SCITXBUF	16 bits	9	SCI Transmit Data Buffer Register
SCIFFTX	16 bits	10	SCI FIFO Transmit Register
SCIFFRX	16 bits	11	SCI FIFO Receive Register
SCIFFCT	16 bits	12	SCI FIFO Control Register
SCIPRI	16 bits	15	SCI Priority Control Register

The code in Example 3 groups the SCI registers together as members of a C/C++ structure. The register in the lowest memory location is listed first in the structure and the register in the highest memory location is listed last. Reserved memory locations are held with variables that are not used except as space holders, for example, rsvd1, rsvd2, rsvd3, and so forth. The register's size is indicated by its type: Uint16 for 16-bit (unsigned int) and Uint32 for 32-bit (unsigned long). The SCI peripheral registers are all 16-bits so only Uint16 has been used.

Example 3. SCI Register-File Structure Definition

```
/********************************************************************
* SCI header file
* Defines a register file structure for the SCI peripheral
********************************************************************/

#define Uint16 unsigned int
#define Uint32 unsigned long

struct  SCI_REGS {
   union  SCICCR_REG    SCICCR;     // Communications control register
   union  SCICTL1_REG   SCICTL1;    // Control register 1
   Uint16               SCIHBAUD;   // Baud rate (high) register
   Uint16               SCILBAUD;   // Baud rate (low) register
   union  SCICTL2_REG   SCICTL2;    // Control register 2
   union  SCIRXST_REG   SCIRXST;    // Receive status register
   Uint16               SCIRXEMU;   // Receive emulation buffer register
   union  SCIRXBUF_REG  SCIRXBUF;   // Receive data buffer
   Uint16               rsvd1;      // reserved
   Uint16               SCITXBUF;   // Transmit data buffer
   union  SCIFFTX_REG   SCIFFTX;    // FIFO transmit register
   union  SCIFFRX_REG   SCIFFRX;    // FIFO receive register
   union  SCIFFCT_REG   SCIFFCT;    // FIFO control register
   Uint16               rsvd2;      // reserved
   Uint16               rsvd3;      // reserved
   union  SCIPRI_REG    SCIPRI;     // FIFO Priority control
};
```

The structure definition in Example 3 creates a new type called *struct SCI_REGS*. The definition alone does not create any variables. Example 4 shows how variables of type struct *SCI_REGS* are created in a way similar to built-in types such as int or unsigned int. Multiple instances of the same peripheral use the same type definition. If there are two SCI peripherals on a device, then two variables are created: *SciaRegs* and *ScibRegs*.

Example 4. SCI Register-File Structure Variables

```
/********************************************************************
* Source file using register-file structures
* Create a variable for each of the SCI register files
********************************************************************/

volatile struct SCI_REGS SciaRegs;
volatile struct SCI_REGS ScibRegs;
```

The volatile keyword is very important in Example 4. A variable is declared as volatile whenever its value can be changed by something outside the control of the code in which it appears. For example, peripheral registers can be changed by the hardware itself or within an interrupt. If volatile is not specified, then it is assumed the variable can only be modified by the code in which it appears and the compiler may optimize out what is seen as an unnecessary access. The compiler will not, however, optimize out any volatile variable access; this is true even if the compiler's optimizer is enabled.

3.2 Using the DATA_SECTION Pragma to Map a Register-File Structure to Memory

The compiler produces relocatable blocks of code and data. These blocks, called sections, are allocated in memory in a variety of ways to conform to different system configurations. The section to memory block assignments are defined in the linker command file.

By default, the compiler assigns global and static variables like *SciaRegs* and *ScibRegs* to the .ebss or .bss section. In the case of the abstraction layer, however, the register-file variables are instead allocated to the same memory as the peripheral's register file. Each variable is assigned to a specific data section outside of .bss/ebss by using the compiler's DATA_SECTION pragma.

The syntax for the DATA_SECTION pragma in C is:

```
#pragma DATA_SECTION (symbol,"section name")
```

The syntax for the DATA_SECTION pragma in C++ is:

```
#pragma DATA_SECTION ("section name")
```

The DATA_SECTION pragma allocates space for the *symbol* in the section called *section name*. In Example 5, the DATA_SECTION pragma is used to assign the variable *SciaRegs* and *ScibRegs* to data sections named *SciaRegsFile* and *ScibRegsFile*. The data sections are then directly mapped to the same memory block occupied by the respective SCI registers.

Example 5. Assigning Variables to Data Sections

```
/********************************************************************
* Assign variables to data sections using the #pragma compiler statement
* C and C++ use different forms of the #pragma statement
* When compiling a C++ program, the compiler will define __cplusplus automatically
********************************************************************/
//----------------------------------------
#ifdef __cplusplus
#pragma DATA_SECTION("SciaRegsFile")
#else
#pragma DATA_SECTION(SciaRegs,"SciaRegsFile");
#endif
volatile struct SCI_REGS SciaRegs;

//----------------------------------------
#ifdef __cplusplus
#pragma DATA_SECTION("ScibRegsFile")
#else
#pragma DATA_SECTION(ScibRegs,"ScibRegsFile");
#endif
volatile struct SCI_REGS ScibRegs;
```

This data section assignment is repeated for each peripheral. The linker command file is then modified to map each data section directly to the memory space where the registers are mapped. For example, Table 1 indicates that the SCI-A registers are memory mapped starting at address 0x7050. Using the assigned data section, the variable *SciaRegs* is allocated to a memory block starting at address 0x7050. The memory allocation is defined in the linker command file (.cmd) as shown in Example 6. For more information on using the C28x linker and linker command files, see the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513).

Example 6. Mapping Data Sections to Register Memory Locations

```
/********************************************************************
* Memory linker .cmd file
* Assign the SCI register-file structures to the corresponding memory
********************************************************************/

MEMORY
{
...
   PAGE 1:
   SCIA        : origin = 0x007050, length = 0x000010     /* SCI-A registers */
   SCIB        : origin = 0x007750, length = 0x000010     /* SCI-B registers */
...
}

SECTIONS
{
...
   SciaRegsFile      : > SCIA,        PAGE = 1
   ScibRegsFile      : > SCIB,        PAGE = 1
...
}
```

By mapping the register-file structure variable directly to the memory address of the peripheral's registers, you can access the registers directly in C/C++ code by simply modifying the required member of the structure. Each member of a structure can be used just like a normal variable, but its name will be a bit longer. For example, to write to the SCI-A Control Register (SCICCR), access the SCICCR member of *SciaRegs* as shown in Example 7. Here the dot is an operator in C that selects a member from a structure.

Example 7. Accessing a Member of the SCI Register-File Structure

```
/********************************************************************
* User's source file
********************************************************************/
...
SciaRegs.SCICCR = SCICCRA_MASK;
ScibRegs.SCICCR = SCICCRB_MASK;
...
```

3.3 Adding Bit-Field Definitions

Accessing specific bits within the register is often useful; bit-field definitions provide this flexibility. Bit fields are defined within a C/C++ structure by providing a list of bit-field names, each followed by colon and the number of bits the field occupies.

Bit fields are a convenient way to express many difficult operations in C or C++. Bit fields do, however, suffer from a lack of portability between hardware platforms. On the C28x devices, the following rules apply to bit fields:

- Bit field members are stored from right to left in memory. That is, the least significant bit, or bit zero, of the register corresponds to the first bit field.
- If the total number of bits defined by bit fields within a structure grows above 16 bits, then the next bit field is stored consecutively in the next word of memory.

The SCICCR and SCICTL1 registers in Figure 1 and Figure 2 translate into the C/C++ bit-field definitions in Example 8. Reserved locations within the register are held with bit fields that are not used except as place holders, i.e., rsvd, rsvd1, rsvd2, et cetera. As with other structures, each member is accessed using the dot operator in C or C++.

Figure 1. SCI SCICCR Register

15	14	13	12	11	10	9	8
Reserved							
R-0							

7	6	5	4	3	2		0
STOPBITS	EVEN/ODD PARITY	PRIORITY ENABLE	LOOPBACK ENA	ADDR/IDLE Mode	SCICHAR		
R-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0		

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure 2. SCI SCICTL1 Register

15							8
Reserved							
R-0							

7	6	5	4	3	2	1	0
Reserved	RXERRINTENA	SWRESET	Reserved	TXWAKE	SLEEP	TXENA	RXENA
R-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Example 8. SCI Control Registers Defined Using Bit Fields

```
/********************************************************************
* SCI header file
********************************************************************/
//----------------------------------------------------------
// SCICCR communication control register bit definitions:
//
struct  SCICCR_BITS {        // bit    description
   Uint16 SCICHAR:3;         // 2:0    Character length control
   Uint16 ADDRIDLE_MODE:1;   // 3      ADDR/IDLE Mode control
   Uint16 LOOPBKENA:1;       // 4      Loop Back enable
   Uint16 PARITYENA:1;       // 5      Parity enable
   Uint16 PARITY:1;          // 6      Even or Odd Parity
   Uint16 STOPBITS:1;        // 7      Number of Stop Bits
   Uint16 rsvd1:8;           // 15:8   reserved
};
//-------------------------------------------
// SCICTL1 control register 1 bit definitions:
//
struct  SCICTL1_BITS {       // bit    description
   Uint16 RXENA:1;           // 0      SCI receiver enable
   Uint16 TXENA:1;           // 1      SCI transmitter enable
   Uint16 SLEEP:1;           // 2      SCI sleep
   Uint16 TXWAKE:1;          // 3      Transmitter wakeup method
   Uint16 rsvd:1;            // 4      reserved
   Uint16 SWRESET:1;         // 5      Software reset
   Uint16 RXERRINTENA:1;     // 6      Receive interrupt enable
   Uint16 rsvd1:9;           // 15:7   reserved

};
```

3.4 Using Unions

While bit fields provide access to individual bits, you may still want to access the register as a single value. To provide this option, a union declaration is created to allow the register to be accessed in terms of the defined bit fields or as a whole. The union definitions for the SCI communications control register and control register 1 are shown in Example 9.

Example 9. Union Definition to Provide Access to Bit Fields and the Whole Register

```
/********************************************************************
* SCI header file
********************************************************************/

union SCICCR_REG {
   Uint16              all;
   struct SCICCR_BITS  bit;
};

union SCICTL1_REG {
   Uint16               all;
   struct SCICTL1_BITS  bit;
};
```

Once bit-field and union definitions are established for specific registers, the SCI register-file structure is rewritten in terms of the union definitions as shown in Example 10. Note that not all registers have bit field definitions; some registers, such as SCITXBUF, will always be accessed as a whole and a bit field definition is not necessary.

Example 10. SCI Register-File Structure Using Unions

```
/********************************************************************
* SCI header file
********************************************************************/

//----------------------------------------------------------------------------
// SCI Register File:
//
struct  SCI_REGS {
   union SCICCR_REG     SCICCR;     // Communications control register
   union SCICTL1_REG    SCICTL1;    // Control register 1
   Uint16               SCIHBAUD;   // Baud rate (high) register
   Uint16               SCILBAUD;   // Baud rate (low) register
   union SCICTL2_REG    SCICTL2;    // Control register 2
   union SCIRXST_REG    SCIRXST;    // Receive status register
   Uint16               SCIRXEMU;   // Receive emulation buffer register
   union SCIRXBUF_REG   SCIRXBUF;   // Receive data buffer
   Uint16               rsvd1;      // reserved
   Uint16               SCITXBUF;   // Transmit data buffer
   union SCIFFTX_REG    SCIFFTX;    // FIFO transmit register
   union SCIFFRX_REG    SCIFFRX;    // FIFO receive register
   union SCIFFCT_REG    SCIFFCT;    // FIFO control register
   Uint16               rsvd2;      // reserved
   Uint16               rsvd3;      // reserved
   union SCIPRI_REG     SCIPRI;     // FIFO Priority control
};
```

As with other structures, each member (.all or .bit) is accessed using the dot operator in C/C++ as shown in Example 11. When the .all member is specified, the entire register is accessed. When the .bit member is specified, then the defined bit fields can be directly accessed.

NOTE: Writing to a bit field has the appearance of writing to only the specified field. In reality, however, the CPU performs what is called a read-modify-write operation; the entire register is read, its contents are modified and the entire value is written back. Possible side effects of read-modify-write instructions are discussed in Section 6.

Example 11. Accessing Bit Fields in C/C++

```
/********************************************************************
* User's source file
********************************************************************/

// Access registers without a bit field definition (.all, .bit not used)
SciaRegs.SCIHBAUD = 0;
SciaRegs.SCILBAUD = 1;

// Write to bit fields in SCI-A SCICTL1
SciaRegs.SCICTL1.bit.SWRESET = 0;
SciaRegs.SCICTL1.bit.SWRESET = 1;
SciaRegs.SCIFFCT.bit.ABDCLR = 1;
SciaRegs.SCIFFCT.bit.CDC = 1;

// Poll (i.e., read) a bit
while(SciaRegs.SCIFFCT.bit.CDC == 1) { }

// Write to the whole SCI-B SCICTL1/2 registers (use .all)
ScibRegs.SCICTL1.all = 0x0003;
ScibRegs.SCICTL2.all = 0x0000;
```

4 Bit Field and Register-File Structure Advantages

The bit field and register-file structure approach has many advantages that include:

- **Register-file structures and bit fields are already available from Texas Instruments.**

  In the C/C++ Header Files and Peripheral Examples, the register-file structures and bit fields have been implemented for all peripherals on the C28x cores of the TMS320x28xx and TMS320x28xxx devices. The included header files can be used as-is or extended to suit your particular needs.

  The complete implementation is available in the software downloads from TI's website as shown in Section 1.

- **Using bit fields produces code that is easy-to-write, easy-to-read, easy-to-update, and efficient.**

  Bit fields can be manipulated quickly without the need to determine a register mask value. In addition, you have the flexibility to access registers either by bit field or as a single quantity as shown in Example 11. Code written using the register file structures also generates very efficient code. Code efficiency will be discussed in Section 5.

- **Bit fields take advantage of the Code Composer Studio editors auto complete feature.**

  At first it may seem that variable names are harder to remember and longer to type when using register-file structures and bit fields. The Code Composer Studio editor provides a list of possible structure/bit field elements as you type; this makes it easier to write code without referring to documentation for register and bit field names. An example of the auto completion feature for the CPU-Timer TCR register is shown in Figure 3.

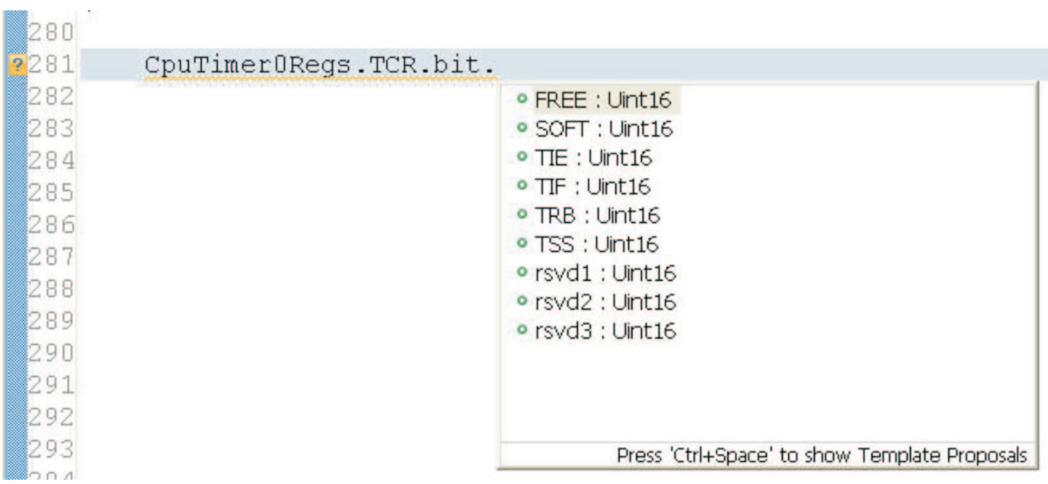

Figure 3. Code Composer Studio v5.1 Autocomplete Feature

- **Increases the effectiveness of the Code Composer Studio Watch Window.**
  You can add and expand register-file structures in Code Composer Studio's watch window as shown in Figure 4. Bit field values are read directly without extracting their value by hand.

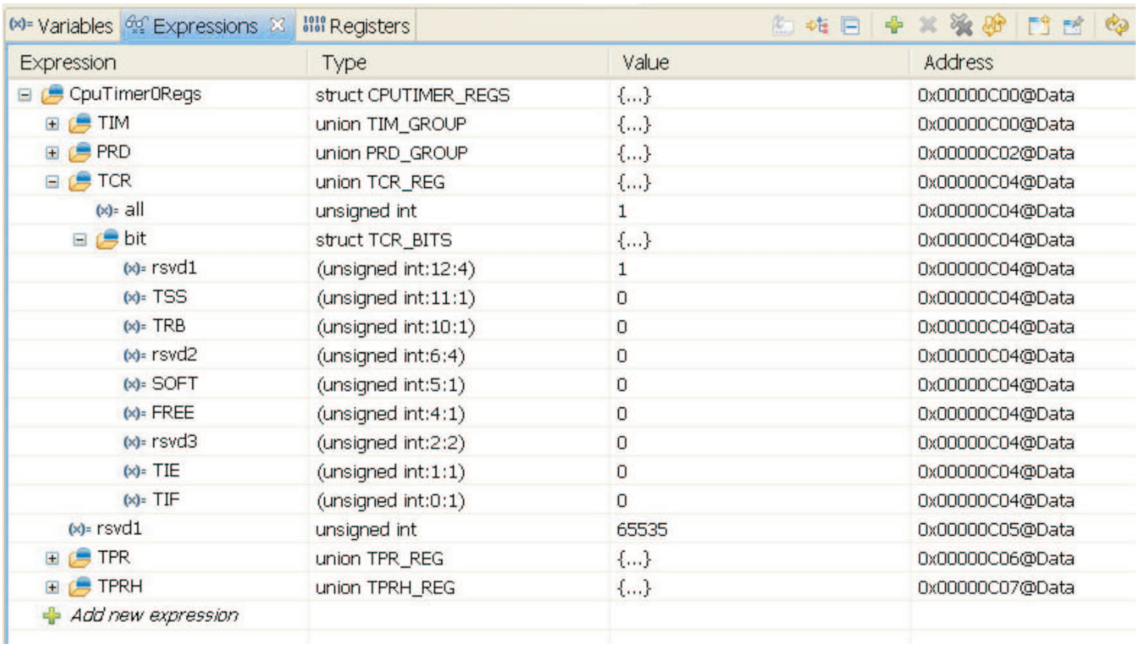

Figure 4. Code Composer Studio v5.1 Expression Window

5 Code Size and Performance Using Bit Fields

The bit field and register-file structure approach is very efficient when accessing a single bit within a register or when polling a bit. As an example, consider code to initialize the PCLKCR0 register on a TMS320x280x device. PCLKCR0 is described in detail in the *TMS320x280x, 2801x, 2804x System Control and Interrupts Reference Guide* (SPRU712). The bit-field definition for this register is shown in Example 12.

Figure 5. Peripheral Clock Control 0 Register (PCLKCR0)

15	14	13	12	11	10	9	8
ECANBENCLK	ECANAENCLK	Reserved		SCIBENCLK	SCIAENCLK	SPIBENCLK	SPIAENCLK
R/W-0	R/W-0	R-0		R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
SPIDENCLK	SPICENCLK	Reserved	I2CAENCLK	ADCENCLK	TBCLKSYNC	Reserved	
R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R-0	

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Example 12. TMS320x280x PCLKCR0 Bit-Field Definition

```
// Peripheral clock control register 0 bit definitions:
struct PCLKCR0_BITS  {  // bits  description
   Uint16 rsvd1:2;      // 1:0   reserved
   Uint16 TBCLKSYNC:1;  // 2     eWPM Module TBCLK enable/sync
   Uint16 ADCENCLK:1;   // 3     Enable high speed clk to ADC
   Uint16 I2CAENCLK:1;  // 4     Enable SYSCLKOUT to I2C-A
   Uint16 rsvd2:1;      // 5     reserved
   Uint16 SPICENCLK:1;  // 6     Enable low speed clk to SPI-C
   Uint16 SPIDENCLK:1;  // 7     Enable low speed clk to SPI-D
   Uint16 SPIAENCLK:1;  // 8     Enable low speed clk to SPI-A
   Uint16 SPIBENCLK:1;  // 9     Enable low speed clk to SPI-B
   Uint16 SCIAENCLK:1;  // 10    Enable low speed clk to SCI-A
   Uint16 SCIBENCLK:1;  // 11    Enable low speed clk to SCI-B
   Uint16 rsvd3:2;      // 13:12 reserved
   Uint16 ECANAENCLK:1; // 14    Enable SYSCLKOUT to eCAN-A
   Uint16 ECANBENCLK:1; // 15    Enable SYSCLKOUT to eCAN-B
};
```

The code in Example 13 enables the peripheral clocks on a TMS320x2801 device. The C28x compiler generates one assembly code instruction for each C-code register access. This is very efficient; there is a one-to-one correlation between the C instructions and the assembly instructions. The only overhead is the initial instruction to set the data page pointer (DP).

Example 13. Assembly Code Generated by Bit Field Accesses

```
   C-Source Code                                       Generated Assembly
                                                       Memory    Instruction

// Enable only 2801 Peripheral Clocks
EALLOW;                                                3F82A7    EALLOW
                                                       3F82A8    MOVW    DP,#0x01C0
SysCtrlRegs.PCLKCR0.bit.rsvd1 =     0;                 3F82AA    AND    @28,#0xFFFC
SysCtrlRegs.PCLKCR0.bit.TBCLKSYNC = 0;                 3F82AC    AND    @28,#0xFFFB
SysCtrlRegs.PCLKCR0.bit.ADCENCLK =  1;                 3F82AE    OR     @28,#0x0008
SysCtrlRegs.PCLKCR0.bit.I2CAENCLK = 1;                 3F82B0    OR     @28,#0x0010
SysCtrlRegs.PCLKCR0.bit.rsvd2 =     0;                 3F82B2    AND    @28,#0xFFDF
SysCtrlRegs.PCLKCR0.bit.SPICENCLK = 1;                 3F82B4    OR     @28,#0x0040
SysCtrlRegs.PCLKCR0.bit.SPIDENCLK = 1;                 3F82B6    OR     @28,#0x0080
SysCtrlRegs.PCLKCR0.bit.SPIAENCLK = 1;                 3F82B8    OR     @28,#0x0100
SysCtrlRegs.PCLKCR0.bit.SPIBENCLK = 1;                 3F82BA    OR     @28,#0x0200
SysCtrlRegs.PCLKCR0.bit.SCIAENCLK = 1;                 3F82BC    OR     @28,#0x0400
SysCtrlRegs.PCLKCR0.bit.SCIBENCLK = 0;                 3F82BE    AND    @28,#0xF7FF
SysCtrlRegs.PCLKCR0.bit.rsvd3 =     0;                 3F82C0    AND    @28,#0xCFFF
SysCtrlRegs.PCLKCR0.bit.ECANAENCLK= 1;                 3F82C2    OR     @28,#0x4000
SysCtrlRegs.PCLKCR0.bit.ECANBENCLK= 0;                 3F82C4    AND    @28,#0x7FFF
EDIS;                                                  3F82C6    EDIS
```

NOTE: EALLOW and EDIS are macros defined in the C/C++ Header Files and Peripheral Examples. These macros expand to the EALLOW and EDIS assembly instructions.

The EALLOW protection mechanism prevents spurious CPU writes to several registers. Executing EALLOW permits the CPU to write freely to protected registers and executing EDIS protects them once more. For information on EALLOW protection and a list of protected registers, see the device-specific *System Control and Interrupts Reference Guide or Technical Reference Manual (TRM)*.

To calculate how many cycles the code in Example 13 will take, you need to know how many wait states are required to access the PCLKCR0 register. Wait state information for all memory blocks and peripheral frames is listed in the device specific data manual. The PCLKCR0 register is in peripheral frame 2; this frame requires two wait states for a read access and no wait states for a write access. This means a read from PCLKCR0 takes three cycles total and a write takes one cycle. In addition, a new access to PCLKCR0 cannot begin until the previous write is complete. This built-in protection mechanism removes pipeline effects and makes sure operations proceed in the correct order; all of the peripheral registers have this protection. In Example 13, each access to the PCLKCR0 register will take six cycles; the pipeline phases are shown in Table 3.

Table 3. CPU-Pipeline Activity For Read-Modify-Write Instructions in Example 13

CPU-Pipeline Phase [(1)]				
Read 1 - Read Begins	**Read 2 - Data Latched**	**Execute - Value Modified**	**Write - Value written**	**Cycle**
`AND @28,#0xFFFC`				1
`AND @28,#0xFFFC`				2
`AND @28,#0xFFFC`				3
	`AND @28,#0xFFFC`			4
		`AND @28,#0xFFFC`		5
			`AND @28,#0xFFFC`	6
`AND @28,#0xFFFB`				7
`AND @28,#0xFFFB`				8
`AND @28,#0xFFFB`				9
	`AND @28,#0xFFFB`			10
		`AND @28,#0xFFFB`		11
			`AND @28,#0xFFFB`	12
`OR @28,#0x0008`				13
`OR @28,#0x0008`				14
`OR @28,#0x0008`				15
	`OR @28,#0x0008`			16
		`OR @28,#0x0008`		17
			`OR @28,#0x0008`	18
`OR @28,#0x0010`				
etc...				

(1) For detailed CPU pipeline information, see the *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430).

When code size and cycle counts must be kept to a minimum, it is beneficial to reduce the number of instructions required to initialize a register to as few as possible. Here are some options for reducing code size:

- **Enable the compiler's optimizer:**

  As mentioned in Section 3.1, register-file variables are declared as volatile. For this reason, enabling the optimizer alone will **not** reduce the number of instructions. The keyword volatile alerts the compiler that the variable's value can change outside of the currently executing code. While removing the volatile keyword would reduce code size, it is not recommended. Removing volatile must be done with great care and only where the developer is certain doing so will not yield incorrect results.

- **Write to the complete register (.all union member):**

  The union definitions discussed in Section 3.4 allow access to either specific bit fields or to the entire register. When a write is performed to the entire register using the .all member of the union, code size is reduced. This method creates very efficient code as shown in Example 14. Using .all, however, makes the code both harder to write and harder to read. It is not immediately evident how different bit fields in the register are configured.

Example 14. Optimization Using the .all Union Member

```
   C-Source Code                                    Generated Assembly
                                                    Memory    Instruction


EALLOW;                                             3F82A7    EALLOW
SysCtrlRegs.PCLKCR0.all = 0x47D8;                   3F82A8    MOVW    DP,#0x01C0
EDIS;                                               3F82AA    MOV    @28,#0x47D8
                                                    3F82AC    EDIS
```

- **Use a shadow register and enable the compiler's optimizer:**

  This method is the best compromise. The register's contents are loaded into a shadow register of the same type as shown in Example 15. The content of the shadow register is then modified using bit fields. Since *shadowPCLKCR0* is not volatile, the compiler will combine the bit field writes when the optimizer is enabled. Note that all of the reserved locations are also initialized. At the end of the code the value in the shadow register is written to PCLKCR0. This method retains the advantages of bit field definitions and results in code that is easy to read. The assembly shown was generated with the compiler's optimization level -o1 enabled.

Example 15. Optimization Using a Shadow Register

```
   C-Source Code                                    Generated Assembly
                                                    Memory    Instruction

// Enable only 2801 Peripheral Clocks
union PCLKCR0_REG  shadowPCLKCR0;
EALLOW;                                             3F82A7    EALLOW
shadowPCLKCR0.bit.rsvd1 =     0;                    3F82A8    MOV  @AL,#0x47D8
shadowPCLKCR0.bit.TBCLKSYNC = 0;                    3F82AA    MOVW  DP,#0x01C0
shadowPCLKCR0.bit.ADCENCLK =  1;   // ADC           3F82AC    MOV  @28,AL
shadowPCLKCR0.bit.I2CAENCLK = 1;   // I2C           3F82AD    EDIS
shadowPCLKCR0.bit.rsvd2 =     0;
shadowPCLKCR0.bit.SPICENCLK = 1;   // SPI-C
shadowPCLKCR0.bit.SPIDENCLK = 1;   // SPI-D
shadowPCLKCR0.bit.SPIAENCLK = 1;   // SPI-A
shadowPCLKCR0.bit.SPIBENCLK = 1;   // SPI-B
shadowPCLKCR0.bit.SCIAENCLK = 1;   // SCI-A
shadowPCLKCR0.bit.SCIBENCLK = 0;   // SCI-B
shadowPCLKCR0.bit.rsvd3 =     0;
shadowPCLKCR0.bit.ECANAENCLK= 1;   // eCAN-A
shadowPCLKCR0.bit.ECANBENCLK= 0;   // eCAN-B
SysCtrlRegs.PCLKCR0.all = shadowPCLKCR0.all;
EDIS;
```

6 Read-Modify-Write Considerations When Using Bit Fields

When writing to a bit field, the compiler generates what is called a read-modify-write assembly instruction. Read-modify-write refers to the technique used to implement bit-wise or byte-wise operations such as AND, OR and XOR. That is, the location is *read*, the single bit field is *modified*, and the result is *written* back. Example 16 shows some of the C28x read-modify-write assembly instructions.

Example 16. A Few Read-Modify-Write Operations

```
AND  @Var, #0xFFFC        ; Read 16-bit value "Var"
                          ; AND the value with 0xFFFC
                          ; Write the 16-bit result to "Var"
                          ;
                          ;
OR   @Var, #0x0010        ; Read 16-bit value "Var"
                          ; OR the value with 0x0010
                          ; Write the 16-bit result to "Var"
                          ;
                          ;
XOR  @VarB, AL            ; Read 16-bit value "Var"
                          ; XOR with AL
                          ; Write the 16-bit result to "Var"
                          ;
                          ;
MOVB *+XAR2[0], AH.LSB    ; Read 16-bit value pointed to by XAR2
                          ; Modify the least significant byte
                          ; Write the 16-bit value back
```

With a full CPU pipeline, a C28x based device can complete one read-modify-write operation to zero wait-state SARAM every cycle. When accessing the peripheral registers or external memory, however, required wait states must be taken into account. In addition, the pipeline protection mechanism can further stall instructions in the CPU pipeline. This is described in more detail in Section 5 and in the *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430).

Read-modify-write instructions usually have no ill side effects. It is important, however, to realize that read-modify-write instructions do not limit access to only specific bits in the register; these instructions write to all of the register's bits. In some cases, the read-modify-write sequence can cause unexpected results when bits are written to with the value originally read. Registers that are sensitive to read-modify-write instructions fall into three categories:

- Registers with bits that hardware can change after the read, but before the write
- Registers with write 1-to-clear bits
- Registers that have bits that must be written with a value different from what the bits read back

Registers that fall into these three categories are typically found within older peripherals. To keep register compatibility, the register files have not been redesigned to avoid this issue. Newer peripherals, such as the ePWM, eCAP, and eQEP, however, have a register layout specifically designed to avoid these problems.

This section describes in detail the three categories in which read-modify-write operations should be used with care. In addition, an example of each type of register is given along with a suggested method for safely modifying that register. At the end of the section a list of read-modify-write sensitive registers is provided for reference.

6.1 Registers That Hardware Can Modify During Read-Modify-Write Operations

The device itself can change the state of some bits between the read and the write stages of the CPU pipeline. For example, the PIE interrupt flag registers (PIEIFRx where x = 1, 2, ... 12) can change due to an external hardware or peripheral event. The value written back may overwrite a flag, corrupting the value, and result in missed interrupts.

6.1.1 PIEIFRx Registers

If there is a need to clear a PIEIFRx bit, then the rule is to always let the CPU take the interrupt to clear the flag. This is done by re-mapping the interrupt vector to a pseudo interrupt service routine (ISR). The corresponding PIEIERx bit is then set to allow the CPU to service the interrupt using the pseudo ISR. Within the pseudo ISR the interrupt vector is re-mapped to the interrupt vector for the true ISR routine as shown in Example 17.

NOTE: This rule does not apply to the CPU's IFR register. Special instructions are provided to clear CPU IFR bits and will not result in missing interrupts. Use the *OR IFR* instruction to set IFR bits, and use the *AND IFR* instruction to clear pending interrupts.

Example 17. Clearing PIEIFRx (x = 1, 2...12) Registers

```
/**********************************************************************
* User's source file
**********************************************************************/

// Pseudo ISR prototype. PTIN = pointer to an interrupt
interrupt void PseudoISR(void);
PINT TempISR;
....
  if( PieCtrlRegs.PIEIFR1.bit.INTx4 == 1)
  {
    // Temp save current vector and remap to pseudo ISR
    // Take the interrupt to clear the PIEIFR flag
    EALLOW;
    TempISR = PieVectTable.XINT1;
    PieVectTable.XINT1 = PseudoISR;
    PieCtrlRegs.PIEIER1.bit.INTx4 = 1;
    EDIS;
  }
....

  // Pseudo ISR
  // Services the interrupt & the hardware clears the PIEIFR flag
  // Re-maps the interrupt to the proper ISR
  interrupt void PseudoISR(void)
  {
    EALLOW;
    PieVectTable.XINT1 = TempISR;
    EDIS;
  }
```

6.1.2 GPxDAT Registers

Another case of bits that can change between a read and a write are the GPIO data registers. Consider the code shown in Example 18. Except on 281x devices, the GPxDAT registers reflect the state of the pin, not the output latch. This means the register reflects the actual pin value. However, there is a lag between when the register is written to when the new pin value is reflected back in the register. This may pose a problem when this register is used in subsequent program statements to alter the state of GPIO pins. In Example 18, two program statements attempt to drive two different GPIO pins. The second instruction will wait for the first to finish its write due to the write-followed-by-read protection on this peripheral frame. There will be some lag, however, between the write of GPIO16 and the GPxDAT bit reflecting the new value (1) on the pin. During this lag, the second instruction will read the old value of GPIO16 (0) and write it back along with the new value of GPIO17 (0). Therefore, the GPIO16 pin stays low.

One solution is to put some NOP's between the read-modify-write instructions. A better solution is to use the GPxSET/GPxCLEAR/GPxTOGGLE registers instead of the GPxDAT registers. These registers always read back a 0 and writes of 0 have no effect. Only bits that need to be changed can be specified without disturbing any other bit(s) that are currently in the process of changing. The same code using GPxSET and GPxCLEAR registers is shown in Example 19.

Example 18. Read-Modify-Write Effects on GPxDAT Registers

```
/********************************************************************
* User's source file
********************************************************************/

for(;;)
{
     // Make LED Green
     GpioDataRegs.GPADAT.bit.GPIO16 = 1; // (1) RED_LED_OFF;
     // Read-modify-write occurs

     GpioDataRegs.GPADAT.bit.GPIO17 = 0; // (2) GREEN_LED_ON;
     // Read:   Because of the delay between output to input
     //         the old value of GPIO16 (zero) is read
     // Modify: Changes GPIO17 to a 0
     // Write:  Writes back GPADAT with GPIO16 = 0 and GPIO17 = 0
     delay_loop();

     // Make LED Red
     GpioDataRegs.GPADAT.bit.GPIO16 = 0; // (3) RED_LED_ON;
     GpioDataRegs.GPADAT.bit.GPIO17 = 1; // (4) GREEN_LED_OFF;
     delay_loop();
}
```

Example 19. Using GPxSET and GPxCLEAR Registers

```
/*********************************************************************
* User's source file
********************************************************************/

for(;;)
{
     // Make LED Green
     GpioDataRegs.GPASET.bit.GPIO16 = 1;   // RED_LED_OFF;
     GpioDataRegs.GPACLEAR.bit.GPIO17 = 1; // GREEN_LED_ON;
     delay_loop();

     // Make LED Red
     GpioDataRegs.GPACLEAR.bit.GPIO16 = 1; // RED_LED_ON;
     GpioDataRegs.GPASET.bit.GPIO17 = 1;   // GREEN_LED_OFF;
     delay_loop();
}
```

6.2 Registers With Write 1-to-Clear Bits.

Some registers have what is called write 1-to-clear bits. This means that when the bit is set it can only be cleared by writing a value of one to the bit. During a read-modify-write operation, if a bit is one when it is read, then it will also be written as a one unless it is changed during the modify portion of the access. For this reason, it is likely a read-modify-write instruction will inadvertently clear a write 1-to-clear bit.

The CPU-Timer interrupt flag (TIF) within the TCR register is an example of a write 1-to-clear bit. TIF can be read to determine if the CPU-Timer has overflowed and flagged an interrupt. Example 20 shows code that stops the CPU-Timer and then checks to see if the interrupt flag is set.

Example 20. Read-Modify-Write Operation Inadvertently Modifies Write 1-to-Clear Bits (TCR[TIF])

```
   C-Source Code                                  Generated Assembly
                                                  Memory    Instruction

// Stop the CPU-Timer
CpuTimer0Regs.TCR.bit.TSS = 1;                    3F80C7    MOVW     DP,#0x0030
                                                  3F80C9    OR       @4,#0x0010
// Check to see if TIF is set                     3F80CB    TBIT     @4,#15
if (CpuTimer0Regs.TCR.bit.TIF == 1)               3F80CC    SBF      L1,NTC
{                                                 3F80CD    NOP
   // TIF set, insert action here                 3F80CE L1:
   // NOP is only a place holder                  ....
   asm("   NOP");
}
```

The test for TIF in Example 20 will never be true even if an interrupt has been flagged. The OR assembly instruction to set the TSS bit performs a read-modify-write operation on the TCR register. If the TIF bit is set when the read-modify-write operation occurs, then TIF will be read as a 1 and also written back as a 1. The TIF bit will always be cleared as a result of this write. To avoid this, the write to TIF bit always be 0. The TIF bit ignores writes of 0, thus, its value will be preserved. One possible implementation that preserves TIF is shown in Example 21.

Example 21. Using a Shadow Register to Preserve Write 1-to-Clear Bits

```
   C-Source Code                                   Generated Assembly
                                                   Memory    Instruction
union TCR_REG shadowTCR;
// Use a shadow register to stop the timer
// and preserve TIF (write 1-to-clear bit)
shadowTCR.all = CpuTimer0Regs.TCR.all;            3F80C7     MOVW     DP,#0x0030
shadowTCR.bit.TSS = 1;                            3F80C9     MOV      AL,@4
shadowTCR.bit.TIF = 0;                            3F80CA     ORB      AL,#0x10
CpuTimer0Regs.TCR.all = shadowTCR.all;            3F80CB     MOVL     XAR5,#0x000C00
                                                  3F80CD     AND      AL,@AL,#0x7FFF
// Check the TIF flag                             3F80CF     MOV      *+XAR5[4],AL
if(CpuTimer0Regs.TCR.bit.TIF == 1)                3F80D0     TBIT     *+XAR5[4],#15
{                                                 3F80D1     SBF      L1,NTC
   // TIF set, insert action here                 3F80D2     NOP
   // NOP is only a place holder                  3F80D3 L1:
   asm("   NOP");
}
```

The content of the TCR register is copied into a shadow register. Within the shadow register the TSS bit is set, and the TIF bit is cleared. The shadow register is then written back to TCR; the timer is stopped and the state of TIF is preserved. The assembly instructions were generated with optimization level -o2 enabled.

6.3 Register Bits Requiring a Specific Value

Some registers have bits that must be written as a specific value. If this value is different from the value the bits read, then a read-modify-write operation will likely write the incorrect value.

An example is the watchdog check bit field (WDCHK) in the watchdog control register. The watchdog check bits must be written as 1,0,1; any other value is considered illegal and will reset the device. Since these bits always read back as 0,0,0, a read-modify-write operation will write 0,0,0 unless WDCHK is changed during the modify portion of the operation.

Another solution is to avoid the read-modify-write operation and instead only write a 16-bit value to the WDCR register. To remind you of this requirement, a bit field definition is not provided for the WDCR register in the C/C++ Header Files and Peripheral Examples. Registers that do not have bit-field nor union definitions are accessed without the .bit or .all designations as shown in Example 22.

Example 22. Watchdog Check Bits (WDCR[WDCHK])

```
/**********************************************************************
* User's source file
**********************************************************************/

SysCtrlRegs.WDCR = 0x0068;
```

See the *TMS320x280x, 2801x, 2804x DSP System Control and Interrupts Reference Guide* (SPRU712) and *TMS320x281x System Control and Interrupts Reference Guide* (SPRU078) for more information on the watchdog module.

6.4 Read-Modify-Write Sensitive Registers

Table 4 lists registers that are sensitive to read-modify-write instructions. Depending on the register and how the peripheral is used in the application, effects of a read-modify-write operation may or may not be a concern. This list may not be complete.

Table 4. Read-Modify-Write Sensitive Registers

Module	Registers		Comments
Watchdog	SCSR		WDOVERRIDE is a write 1-to-clear bit and always reads back as a 1.
	WDCR		WDCHK must be written as 1,0,1 and always read back as 0,0,0.
	WDCR		WDFLG is a write 1-to-clear bit.
CPU-Timer	TCR		Timer interrupt flag (TIF) is a write 1-to-clear bit.
GPIO	GPxDAT		Use this register to read data and instead use the SET/CLEAR and TOGGLE registers to change the state of GPIO pins.
PIE	PIEIFRx		To clear PIEIFR bits, do not write to the PIEIFR register. Instead map the interrupt to a "pseudo" interrupt and service it. That is, let the hardware clear the interrupt flag otherwise interrupts from other peripherals may be missed.
	PIEACKx		The PIEACK bits are write 1-to-clear bits.
Event Manager (EV)[(1)]	CAPCONA	CAPCONB	CAPRES is a write-0-to-reset bit and always reads back as 0.
	CAPFIFOA	CAPFIFOB	If a write occurs at the same time that a CAPxFIFO status bit is being updated, the write data takes precedence. Thus if the bit changes between the read and the write phase of a read-modify-write instruction, the new bit value may be lost.
	EVAIFRA/B/C	EVBIFRA/B/C	The EV interrupt flags are all write 1-to-clear bits.
eCAN	CANTRS	CANTRR	The eCAN module can change the state of a bit between the time the register is read and the time it is written back.
	CANTA CANRMP CANRFP CANGIF0 CANTOS	CANAA CANRML CANES CANGIF1	These registers contain one or more write 1-to-clear bits.
SPI	SPIST		Contains write 1-to-clear bits.
I²C	I2CSTR		Contains write 1-to-clear bits.

(1) The EV and eCAN modules are not available on all devices.

7 Special Case Peripherals

Access to peripherals occur on one of three peripheral frames (or busses). Peripheral registers are located in the frame capable of accesses that best fit the register set.

- **Peripheral frame 0:**

  Peripherals within this frame are on the device's memory bus. This bus is capable of both 16-bit or 32-bit accesses. For example, CPU-Timers are on the memory bus.
- **Peripheral frame 1:**

  Peripheral frame 1 uses a bus that is capable of both 16-bit and 32-bit accesses. Examples include the ePWM and eCAN peripherals.
- **Peripheral frame 2:**

  Peripheral frame 2 uses a bus that is capable of only 16-bit accesses. All of the peripheral registers on frame 2 are only 16-bits in length. Examples include the SCI, SPI, ADC and I2C.

7.1 eCAN Control Registers

The eCAN control and status registers are limited to 32-bit-wide accesses. Accesses of only 16 bits can yield unpredictable results. The eCAN control and status registers must be handled as a special case; they are the only peripheral frame 1 registers limited to 32-bit wide accesses.

Often the compiler will reduce an access to 16-bits if it will save code size or improve performance. Care must be taken to make sure what appears to be a 32-bit access to the eCAN control and status registers is not simplified to a 16-bit access by the compiler. For example, the compiler has reduced the access shown in Example 23 to a 16-bit access to half of the CANMC register.

Example 23. Invalid eCAN Control Register 16-Bit Write

```
   C-Source Code                                    Generated Assembly
                                                    Memory    Instruction

// The compiler will simplify this to               3F81FA    EALLOW
// a 16-bit read-modify-write                       3F81FB    MOVW     DP,#0x0180
EALLOW;                                             3F81FD    OR      @20,#0x2000
ECanaRegs.CANMC.bit.SCB = 1;                        3F81FF    EDIS
EDIS;
```

To force 32-bit accesses, the bit-field definitions and read-modify-write operations must not be used. The register must be read and written using the *.all* member of the union definition and all 32-bits must be read or written.

Unfortunately, not using bit fields or read-modify-write operations reduces the code readability. One solution is to read the entire register into a shadow register, manipulate the value, and then write the new 32-bit value to the register using *.all*. The code in Example 24 uses a shadow register to force a 32-bit access. If more then one register is going to be accessed, then the whole eCAN register file can be shadowed (i.e., struct ECAN_REGS shadowECanaRegs;).

Example 24. Using a Shadow Register to Force a 32-Bit Access

```
   C-Source Code                                    Generated Assembly
                                                    Memory    Instruction
// Use a shadow register to force a
// 32-bit access
union CANMC_REG shadowCANMC;
EALLOW;                                             3F81FA    EALLOW
                                                    3F81FB    MOVW     DP,#0x0180
// 32-bit read of CANMC                             3F81FD    MOVL     ACC,@20
shadowCANMC.all = ECanaRegs.CANMC.all;              3F81FE    OR       @AL,#0x2000
shadowCANMC.bit.SCB = 1;                            3F8200    MOVL     @20,ACC
                                                    3F8201    EDIS
// 32-bit write of CANMC
ECanaRegs.CANMC.all = shadowCANMC.all;
EDIS;
```

7.2 Byte Peripheral Registers

There are some peripherals that require 8-bit byte accesses. To accomplish this, they have been placed on a bridge that allows the peripherals to be accessed as if they are byte-addressable. Peripherals on this bridge are listed in Table 5.

Table 5. Byte Peripherals

Module	Devices
CAN	28004x, 2807x, 2837xS, 2837xD
DCC	28004x
LIN	28004x
USB	2807x, 2837xS, 2837xD

Since the peripheral registers behave in a byte-addressable way, the addresses of the 32-bit memory-mapped registers are placed at address offset increments of 4 (as in 4 8-bit bytes) instead of 2 as they normally would be on a word-addressable peripheral. 16-bit words are offset at increments of 2 instead of 1. Often this can lead to issues with the compiler.

For example, Example 25 shows code that writes to the CAN_IF1CMD register on the F2837xD CAN-A module using bit-field header files defined in the usual manner. The CAN_IF1CMD is located at address 0x048100, but the code below is accessing 0x0480D4 since the code generation tools do not comprehend that the peripheral bridge treats addresses as byte addresses. Also note that the access to TXRQST, which is in the upper word of the register, should be at an offset of +2 that of CAN_IF1CMD.

Example 25. Invalid Byte Peripheral Register Access

```
   C-Source Code                               Generated Assembly
                                               Instruction

// Set Direction to write and set              MOVB         AL, #0x0
// DATA-A/DATA-B to be transferred to          MOVB         AH, #0x83
// message object                              MOVW         DP, #0x1203
CanaRegs.CAN_IF1CMD.all = 0x830000;            MOVL         @0x14, ACC
                                               OR           @0x15, #0x0004
// Set Tx Request Bit
CanaRegs.CAN_IF1CMD.bit.TXRQST = 1;
```

Fortunately, features have been added to the compiler in version 16.6.0.STS to properly handle these alignment differences. The header files for byte peripherals in C2000Ware use a “byte_peripheral” type attribute to generate the correct code. For more details about the attribute, see the *TMS320C28x Optimizing C/C++ Compiler User's Guide*. Example 26 shows the corrected code generated with the “byte_peripheral” type attribute.

Example 26. Byte Peripheral Register Access Using “byte_peripheral” Attribute

```
   C-Source Code                               Generated Assembly
                                               Instruction

// Set Direction to write and set              MOVB         AL, #0x0
// DATA-A/DATA-B to be transferred to          MOVB         AH, #0x83
// message object                              MOVL         XAR4, #0x048100
CanaRegs.CAN_IF1CMD.all = 0x830000;            MOVL         *+XAR4[0], ACC
                                               MOVL         ACC, *+XAR4[0]
// Set Tx Request Bit                          ORB          AH, #0x4
CanaRegs.CAN_IF1CMD.bit.TXRQST = 1;            MOVL         *+XAR4[0], ACC
```

8 C2000 Peripheral Driver Library Approach

The C2000 Peripheral Driver Library (or Driverlib) is a set of low-level drivers for configuring memory-mapped peripheral registers. The Driverlib is a more readable and portable approach than performing direct register accesses either by bit fields or the #define approach.

The Driverlib is written in C and all source code is found within C2000Ware. It provides drivers for all peripherals and provides access to almost all functionality.

The following sections describe how to use the Driverlib and how it is architected.

8.1 Using the Peripheral Driver Library

The Driverlib provides an interface to configure peripherals. This interface is made up of functions and datatypes and #defines that are intended to be used as parameters to those functions.

Every function has been documented in detail, explaining the purpose of the function, how to use it, the meaning of the return value (if not void), and the values that are valid for each parameter. It is important to read this documentation to look for possible usage notes. For example, the function ADC_enableConverter() advises that a delay is required between calling the function and beginning sampling to allow the ADC time to power up. This documentation is found in both PDF and HTML formats in C2000Ware and also in the driver library header files.

Functions for most peripherals will take a base address as their first parameter to indicate which instance of a peripheral is to be configured (for example, SCI-A or SCI-B); the exceptions to this are the peripherals where there is only one instance per core like system control or the PIE. #defines are provided for base addresses of every peripheral instance in a header file called hw_memmap.h. Again using SCI as an example, this is shown in Example 27.

Example 27. SCI-A Driverlib Function Prototype

```
//////////////////////////////////////////////////////////////////////
//
// Snippet from hw_memmap.h showing base address #defines
//
//////////////////////////////////////////////////////////////////////
...
#define SCIA_BASE                   0x00007050U // SCI A Registers
#define SCIB_BASE                   0x00007750U // SCI B Registers
...

//////////////////////////////////////////////////////////////////////
//
// Snippet from sci.h showing API description and base parameter
//
//////////////////////////////////////////////////////////////////////
//
//! Sets the FIFO interrupt level at which interrupts are generated.
//!
//! \param base is the base address of the SCI port.
//!
//! \param txLevel is the transmit FIFO interrupt level, specified as
//! one of the following:
//! SCI_FIFO_TX0, SCI_FIFO_TX1, SCI_FIFO_TX2, ... or SCI_FIFO_TX16.
//!
//! \param rxLevel is the receive FIFO interrupt level, specified as one
//! of the following:
//! SCI_FIFO_RX0, SCI_FIFO_RX1, SCI_FIFO_RX2, ... or SCI_FIFO_RX16.
//!
//! This function sets the FIFO level at which transmit and receive
//! interrupts are generated.
//!
//! \return None.
//
```

Example 27. SCI-A Driverlib Function Prototype (continued)

```
static inline void
SCI_setFIFOInterruptLevel(uint32_t base, SCI_TxFIFOLevel txLevel,
                          SCI_RxFIFOLevel rxLevel)
```

For the other parameters, #defines or enumerated types are often supplied to provide a readable way to specify the desired value. Typically #defines are used when a parameter is a uint32_t or uint16_t and able to take a bitwise OR of several #defined values. Enumerated types are used when only a single value is applicable.

These values determine what is written to the peripheral registers to configure the peripheral. The function will determine which register or registers to write to and what value to write. The function will also perform any necessary EALLOW or EDIS instructions. Since these details are hidden by the functions, it is not required for the user to have complete knowledge of the hardware to program a peripheral. Example 28 shows code that could be found in a user application that demonstrates this; given a source clock rate and a desired baud rate, the function calculates the necessary prescalers and writes them to the appropriate registers.

Example 28. SCI-A Configuration Using the Driverlib

```
//////////////////////////////////////////////////////////////////////
//
// User's source file
//
//////////////////////////////////////////////////////////////////////

//
// Configure SCI-A with a baud rate of 9600, 8-bit data, one stop bit,
// and no parity
//
SCI_setConfig(SCIA_BASE, 25000000, 9600, (SCI_CONFIG_WLEN_8 |
                                          SCI_CONFIG_STOP_ONE |
                                          SCI_CONFIG_PAR_NONE));

//
// Set the FIFO interrupt level to 8 characters for both FIFOs
//
SCI_setFIFOInterruptLevel(SCIA_BASE, SCI_FIFO_TX8, SCI_FIFO_RX8)


//
// While the transmit FIFO is not full, write 0x00
//
while(SCI_getTxFIFOStatus(SCIA_BASE) != SCI_FIFO_TX16)
{
    SCI_writeCharNonBlocking(SCIA_BASE, 0x00);
}
```

Since API names are primarily made up of full English words and a very limited set of acronyms, the actions of much of the code can be easily understood, even with limited commenting. This is not always the case with code that writes directly to registers which typically have short names made up of acronyms or abbreviations.

8.2 Construction of a Driver Library Function

It is useful to understand how Driverlib functions are constructed when debugging or when wanting to access a register or field that is not configurable using existing Driverlib functions. Driverlib uses an approach similar to the traditional #define approach discussed in Section 2 to perform its register accesses. A header file is generated for each peripheral, containing register address offsets and bit masks and shift amounts for each field within those registers. The naming convention they follow is:

- Values that contain an _O_ are register address offsets used to access the value of a register. For example, SCI_O_CCR is used to access the SCICCR register in a SCI module. These can be added to the base address values to get the register address.
- Values that end in _M represent the mask for a multi-bit field in a register. For example, SCI_CCR_SCICHAR_M is a mask for the SCICHAR field in the SCICCR register. Note that fields that are the whole width of the register are not given masks.
- Values that end in _S represent the number of bits to shift a value in order to align it with a multi-bit field. These values match the macro with the same base name but ending with _M.
- All others are single-bit field masks. For example, SCI_CCR_LOOPBKENA corresponds to the LOOPBKENA bit in the SCICCR register.

A sample of the peripheral register header file is shown in Example 29.

Example 29. SCI Register Description Header File (hw_sci.h)

```
//////////////////////////////////////////////////////////////////////
//
// Example register #defines from hw_sci.h
//
//////////////////////////////////////////////////////////////////////

//*******************************************************************
//
// The following are defines for the SCI register offsets
//
//*******************************************************************
#define SCI_O_CCR                 0x0U     // Communications control
#define SCI_O_CTL1                0x1U     // Control register 1
#define SCI_O_HBAUD               0x2U     // Baud rate (high)
#define SCI_O_LBAUD               0x3U     // Baud rate (low)
#define SCI_O_CTL2                0x4U     // Control register 2
#define SCI_O_RXST                0x5U     // Receive status
#define SCI_O_RXEMU               0x6U     // Receive emulation buffer
#define SCI_O_RXBUF               0x7U     // Receive data buffer
#define SCI_O_TXBUF               0x9U     // Transmit data buffer
#define SCI_O_FFTX                0xAU     // FIFO transmit register
#define SCI_O_FFRX                0xBU     // FIFO receive register
#define SCI_O_FFCT                0xCU     // FIFO control register
#define SCI_O_PRI                 0xFU     // SCI Priority control

//*******************************************************************
//
// The following are defines for the bit fields in the SCICCR register
//
//*******************************************************************
#define SCI_CCR_SCICHAR_S         0U
#define SCI_CCR_SCICHAR_M         0x7U     // Character length control
#define SCI_CCR_ADDRIDLE_MODE     0x8U     // ADDR/IDLE Mode control
#define SCI_CCR_LOOPBKENA         0x10U    // Loop Back enable
#define SCI_CCR_PARITYENA         0x20U    // Parity enable
#define SCI_CCR_PARITY            0x40U    // Even or Odd Parity
#define SCI_CCR_STOPBITS          0x80U    // Number of Stop Bits
...
```

These #defines are used in combination with a set of “HWREG(x)” macros defined in hw_types.h where x is the address of the of the memory location to be accessed

- HWREG(x) is used for 32-bit accesses, such as reading a value from a 32-bit counter register.
- HWREGH(x) is used for 16-bit accesses. This can be used to access a 16-bit register or the upper or lower words of a 32-bit register. This is usually the most efficient macro to use.
- HWREGB(x) is used for 8-bit accesses using the __byte() intrinsic. For more information, see the *TMS320C28x Optimizing C/C++ Compiler User's Guide*. It typically should only be used when an 8-bit access is required by the hardware. Otherwise, use HWREGH() and mask and shift out the unwanted bits.
- HWREG_BP(x) is another macro used for 32-bit accesses, but it uses the __byte_peripheral_32() compiler intrinsic. It is meant to work with the byte peripherals described in Section 7. It tells the compiler that the 32-bit access may not be split into two 16-bit read-modify-write operations since the upper word is not at the expected address offset on a byte peripheral.

These macros used in combination with the register description and base address #defines make up the majority of Driverlib code. Example 30 shows how they are used to implement the SCI_setConfig() function.

Example 30. SCI Function Implementation

```
//////////////////////////////////////////////////////////////////////
//
// Example function implementation from Driverlib sci.c
//
//////////////////////////////////////////////////////////////////////

//*********************************************************************
//
// SCI_setConfig
//
//*********************************************************************
void SCI_setConfig(uint32_t base, uint32_t lspclkHz, uint32_t baud,
                   uint32_t config)
{

...
    //
    // Compute the baud rate divider.
    //
    divider = ((lspclkHz  / (baud * 8U)) - 1U);

    //
    // Set the baud rate.
    //
    HWREGH(base + SCI_O_HBAUD) = (divider & 0xFF00U) >> 8U;
    HWREGH(base + SCI_O_LBAUD) = divider & 0x00FFU;

    //
    // Set parity, data length, and number of stop bits.
    //
    HWREGH(base + SCI_O_CCR) = ((HWREGH(base + SCI_O_CCR) &
                                 ~(SCI_CCR_SCICHAR_M |
                                   SCI_CCR_PARITYENA |
                                   SCI_CCR_PARITY |
                                   SCI_CCR_STOPBITS)) | config);
...
}
```

8.3 Peripheral Driver Library Advantages

The peripheral driver library has many advantages, including:

- **Drivers and header files are already available from Texas Instruments.**

  Driverlib drivers, header files, and example projects are available in C2000Ware. All source code is provided, so drivers can be used as-is or extended to suit your particular needs.

  Ffor information on where to download C2000Ware and the devices for which Driverlib is available, see Section 1.

- **Using Driverlib produces code that is easy-to-write and easy-to-read.**

  Since Driverlib abstracts from the actual register accesses that are occurring, a less detailed knowledge of the hardware is required to write an application. For example, the read-modify-write considerations discussed in Section 6 are often not a concern when using Driverlib because the driver implementation handles them.

  This also means that slight differences in hardware across C2000 devices are abstracted, allowing code to be ported more easily. Additionally, the Driverlib is written with readability in mind, so function names and parameter values are descriptive of their functionality.

- **Driverlib has built in debugging features.**

  Many driver functions contain some manner of argument checking. The use of enumerated types provides compile-time argument checking for some parameters. For other parameters, a run-time assert can check the validity of the values passed to the function. When not debugging, the asserts can be turned off, removing the performance overhead.

- **Driverlib is written to optimize well.**

  Driverlib performance and the features used to generate efficient code are discussed in detail in Section 10.

- **The Driverlib has undergone MISRA-C:2012 static analysis.**

  The drivers are compliant with the C2000 MISRA-C:2012 Policy. Details of the policy can be found in *C2000™ MISRA-C Policy*.

9 Code Size and Performance Using Driverlib

In general, software abstraction can come at the cost of performance. However, Driverlib's low level of abstraction and optimization-conscious design make it efficient.

One of the major optimization-friendly features of Driverlib is that most functions have been declared as inline functions. Inlining allows the compiler to treat the functions like macros when the optimizer is turned on (when the compiler option --opt_level is set to 0 or higher). This removes the overhead of the function call and speeds up code execution.

Example 31 shows code that reads ADC conversion results using the inlined ADC_readResult() function with an optimization level of –o2. A single MOV instruction is generated for each function call. This same code when compiled with an optimization level –o2 but inlining turned off (--disable_inlining) generates 22 words of code (4 for ADC_readResult() and 18 in the calling function) and takes 53 cycles to execute.

Example 31. Inlined ADC_readResult() Function Calls

```
C-Source Code                          Generated Assembly
                                       Instruction

tmp[0]=ADC_readResult(ADCARESULT_BASE,    MOV     *-SP[3], *(0:0x0b00)
                      ADC_SOC_NUMBER0);
tmp[1]=ADC_readResult(ADCARESULT_BASE,    MOV     *-SP[2], *(0:0x0b01)
                      ADC_SOC_NUMBER1);
tmp[2]=ADC_readResult(ADCARESULT_BASE,    MOV     *-SP[1], *(0:0x0b02)
                      ADC_SOC_NUMBER2);
```

In addition to removing the overhead of the function call, inlining Driverlib functions can allow the compiler to evaluate some of code at compile time, resulting in smaller, faster code. This is especially true when constants are passed as parameters to the functions.

Example 32 shows the implementation of the ADC_setupSOC() Driverlib function. The function calculates the address to which it needs to write based on the base and socNumber parameters. All other parameters need to be shifted, adjusted, and combined before they can be written to the register.

Example 32 shows the assembly that is generated when the function is inlined and passed constants. Note that all calculations have been performed at compile time and all that remains to be done is the access protection and register write.

Example 32. ADC Function Implementation to be Optimized

```
/////////////////////////////////////////////////////////////////////
//
// Example function implementation from Driverlib adc.h
//
/////////////////////////////////////////////////////////////////////

static inline void
ADC_setupSOC(uint32_t base, ADC_SOCNumber socNumber,
             ADC_Trigger trigger, ADC_Channel channel,
             uint32_t sampleWindow)
{
    uint32_t ctlRegAddr;

...

    // Calculate address for the SOC control register.
    ctlRegAddr = base + ADC_SOCxCTL_OFFSET_BASE +
                 ((uint32_t)socNumber * 2U);

    // Set the configuration of the specified SOC.
    EALLOW;
    HWREG(ctlRegAddr) = ((uint32_t)channel << ADC_SOC0CTL_CHSEL_S) |
                        ((uint32_t)trigger << ADC_SOC0CTL_TRIGSEL_S) |
                        (sampleWindow - 1U);
    EDIS;
}
```

Example 33. Inlined ADC_setupSOC() Function Call

```
   C-Source Code                              Generated Assembly
                                              Instruction

ADC_setupSOC(ADCA_BASE,                       EALLOW
             ADC_SOC_NUMBER0,                 MOVB     AL, #0xf
             ADC_TRIGGER_EPWM1_SOCA,          MOVB     AH, #0x50
             ADC_CH_ADCIN0, 16);              MOVL     XAR4, #0x007410
                                              MOVL     *+XAR4[0], ACC
                                              EDIS
```

10 Comparing and Combining Approaches

The bit field and register-file structure headers and the peripheral driverlib library approaches are compatible and can be used in the same application or independently. This section compares the two approaches and provides guidance on where one may be preferable to the other if a combined approach is used.

One of the key reasons for this is the ability of the compiler to use the data page pointer on bit-field code. Example 34 shows an example of both approaches configuring a CPU Timer and the corresponding assembly below. The assembly shown was generated with optimization level -o2 enabled and Driverlib ASSERTs turned off. The use of the data page pointer means that the bit-field code generated smaller, faster code in this instance. However, the Driverlib code is easier to read and handles the separate pre-scale registers seamlessly.

Example 34. CPU Timer Bit-Field (Left) and Driverlib (Right) Disassembly Comparison

```
// Initialize timer period
CpuTimer0Regs.PRD.all = 10000000;

// Pre-scale to divide by 1
CpuTimer0Regs.TPR.all  = 0;
CpuTimer0Regs.TPRH.all  = 0;

// Reload timer
CpuTimer0Regs.TCR.bit.TRB = 1;
```

```
// Initialize timer period
CPUTimer_setPeriod(
        CPUTIMER0_BASE, 10000000);

// Pre-scale to divide by 1
CPUTimer_setPreScaler(
        CPUTIMER0_BASE, 0);

// Reload timer
CPUTimer_reloadTimerCounter(
        CPUTIMER0_BASE);
```

```
MOV          @AL, #0x9680
MOVB         AH, #0x98
MOVW         DP, #0x30
MOVL         @0x2, ACC
MOV          @0x6, #0
MOV          @0x7, #0
OR           @0x4, #0x0020
```

```
MOV          @AL, #0x9680
MOVB         AH, #0x98
MOVL         XAR4, #0x000c02
MOVB         XAR7, #0x00
MOVB         XAR6, #0x00
MOVL         *+XAR4[0], ACC
MOV          *(0:0x0c07), @AR7
MOV          *(0:0x0c06), @AR6
MOV          @AL, *(0:0x0c04)
ORB          AL, #0x20
MOV          *(0:0x0c04), @AL
```

Figure 6. CPU Timer Bit-Field (Left) and Driverlib (Right) Disassembly Comparison

In Example 34 each single line of bit-field code corresponded to a Driverlib function. This is not always the case; Example 32 shows the ADC_setupSOC() function which configures multiple fields within the ADCSOC0CTL register at once. Another item of note in Example 35 is that the EALLOW and EDIS instructions are used in all of the Driverlib functions to disable and re-enable write protection on the necessary registers. The compiler is able to optimize out the back-to-back EDIS-EALLOW pairs that this results in when the functions are inlined.

Example 35. ADC Bit-Field (Left) and Driverlib (Right) Disassembly Comparison

```
EALLOW;

// SOC0 to convert pin A0.
// EPWM1SOCA as trigger. Sampling
// window of 16.
AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0;
AdcaRegs.ADCSOC0CTL.bit.ACQPS = 15;
AdcaRegs.ADCSOC0CTL.bit.TRIGSEL= 5;

// Use interrupt 1 flag. Enable and
// clear interrupt.
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 0;
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;

EDIS;
```

```
// SOC0 to convert pin A0.
// EPWM1SOCA as trigger. Sampling
// window of 16.
ADC_setupSOC(ADCA_BASE,
             ADC_SOC_NUMBER0,
             ADC_TRIGGER_EPWM1_SOCA,
             ADC_CH_ADCIN0, 16);

// Use interrupt 1 flag. Enable and
// clear interrupt.
ADC_setInterruptSource(
        ADCA_BASE, ADC_INT_NUMBER1,
        ADC_SOC_NUMBER0);
ADC_enableInterrupt(
        ADCA_BASE, ADC_INT_NUMBER1);
ADC_clearInterruptStatus(
        ADCA_BASE, ADC_INT_NUMBER1);
```

```
EALLOW
MOVW        DP, #0x1d0
AND         @0x10, #0x7fff
AND         @0x11, #0xfff8
AND         AL, @0x10, #0xfe00
ORB         AL, #0xf
MOV         @0x10, AL
AND         AL, @0x11, #0xfe0f
ORB         AL, #0x50
MOV         @0x11, AL
AND         @0x7, #0xfff0
OR          @0x7, #0x0020
OR          @0x4, #0x0001
EDIS
```

```
EALLOW
MOVB        AL, #0xf
MOVB        AH, #0x50
MOVL        XAR4, #0x007410
MOVL        *+XAR4[0], ACC
MOV         @AL, *(0:0x7407)
AND         @AL, #0xfff0
MOV         *(0:0x7407), @AL
MOV         @AL, *(0:0x7407)
ORB         AL, #0x20
MOV         *(0:0x7407), @AL
EDIS
```

Figure 7. ADC Bit-Field (Left) and Driverlib (Right) Disassembly Comparison

If using both approaches in one application, here are some considerations on when you may choose one over the other:

- A less detailed understanding of the hardware is required when using Driverlib which makes it a good choice for quickly developing an application. Driverlib is the recommended approach for new applications.
- When porting legacy code from an older C2000 device to a newer, you can continue using bit field and register-file structures. Bit-field headers have been available for several generations of C2000 devices, and for many peripherals, they have remained mostly compatible.
- Use the bit field and register-file structure approach particularly for performance critical code when Driverlib does not meet requirements. In general, the bit-field approach will generate smaller, faster code when repeated accesses are made to the same data page.

11 References

The following references include additional information on topics found in this application report:

- *C281x C/C++ Header Files and Peripheral Examples*
- *C280x, C2801x C/C++ Header Files and Peripheral Examples*
- *C2804x C/C++ Header Files and Peripheral Examples*
- *TMS320C28x CPU and Instruction Set Reference Guide*
- *TMS320C28x Optimizing C/C++ Compiler User's Guide*
- *TMS320C28x Assembly Language Tools User's Guide*
- *TMS320x281x System Control and Interrupts Reference Guide*
- *TMS320x280x, 2801x, 2804x DSP System Control and Interrupts Reference Guide*
- *TMS320x281x Serial Communications Interface (SCI) Reference Guide*
- *C2000™ MISRA-C Policy*

For peripheral guides specific to your device, see *TMS320x28xx, 28xxx DSP Peripherals Reference Guide*

Support for all new microcontrollers is available in the device support section of C2000Ware.

An *Introduction to Texas Instruments C2000 Microcontrollers* has been contributed to the TI Embedded Processors Wiki located at: http://processors.wiki.ti.com/index.php/Category:C2000.

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from D Revision (January 2013) to E Revision **Page**

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265