Application Report
SPRAA10–January 2006

Migrating from CSL 2.x to CSL 3.x

Michelle Richardson — *DSP Software Applications*

ABSTRACT

This application report introduces the changes that have occurred between CSL 2.x and CSL 3.x, and assists users to migrate their application to the CSL 3.x architecture.

Contents

List of Figures

List of Tables

Trademarks

OMAP, C54x, C55x, BIOS are trademarks of Texas Instruments.

1 Introduction

The Chip Support Libraries for new SOC platforms, such as C64x+, C67x+ and OMAP™ devices, follow the CSL 3.x architecture. Earlier devices are still supported with the CSL 2.x architecture. This application report introduces the changes that have occurred between CSL 2.x and CSL 3.x.

1.1 Product Support

The following table shows the version of CSL supported for various TI DSPs.

Table 1. CSL Support Matrix

TMS320 Family	Devices	CSL 2.x	CSL 3.x	Product #
C54x™	All	x		SPRC132
C55x™	All	x		SPRC133
OMAP™	OMAP5912		x	SPRC199
C62x	All	x		SPRC090
C671x	All	x		SPRC090
C641x	All	x		SPRC090
DM64x	All	x		SPRC090
C64x+	C6445		x	SPRC234
C67x+	C672x		x	SPRC223

Please see the CSL Reference Guide for your specific product for a summary of supported APIs for each module.

1.2 CSL 3.x Benefits

The changes in CSL 3.x were designed to improve the earlier version in several specific areas.

Table 2. CSL 3.x Benefits

Category	Requirements	Solution
Portability	Portable across devices Address increasing device complexity Portable across operating systems	Standard APIs with consistent style Layered hardware abstraction OS-independent. Decoupled from DSP/BIOS Applicable to GPP/DSP/Heterogeneous
Macros	Reduce number and redundancy Reduce nesting levels Create a consistent naming convention	Single set of generic macros Minimize nesting (3 levels maximum) Consistent naming convention
Quality	Improve product quality and out-of-box experience	Standardized APIs facilitate review and testability Naming conventions consistent with data sheet
Memory	Flexible memory placement Reduce code Size Make ROM'able	Handle-based accesses No CSL-internal memory allocation Modular organization
Ease-of-Use	Enable easier debug Provide version check and error reporting	Provide structures that are "watchable" in debug watch window Consistent naming conventions to ease learning curve Two-layered architecture Well-defined error status codes returned by APIs
Versioning	Allow application to check the CSL version	Version ID defined in csl_version.h is readable as a value and as a string

2 Overview of CSL 3.x Architecture

CSL 3.x is partitioned into two distinct layers -- the register layer and the functional layer.

The CSL 3.x functional layer provides a set of standard APIs that provide abstraction from hardware. A core set of APIs are common across peripherals, which maintains a similar look and feel across peripherals and devices. In some cases, one or more additional specialized APIs may be implemented for a peripheral, for ease of use. Because of its unique requirements, the interrupt controller (INTC) module is one example of a module with extensions to the standard APIs.

The CSL 3.x register layer provides register and bit field definitions for a peripheral, and macros for manipulating them. It incorporates a register overlay structure that is consistent with the data sheet in naming convention, and register order and placement. This layer also provides a set of common macros for use across peripherals. CSL 3.x centralizes all register-layer macros into a single header file.

Although CSL 2.x has a hardware abstraction layer (HAL) for register bit field manipulation, it is not clearly separated from functional APIs. Unique HAL macros are provided for each peripheral module.

Please see the CSL API Reference Guide for a more detailed overview of the architecture, APIs, macros, structures and symbols.

2.1 Linkage to DSP/BIOS™

CSL 2.x had built-in coupling with DSP/BIOS for the purpose of issuing system ticks to DSP/BIOS using the on-chip timer, and for managing the hardware-interrupt dispatch table.

CSL 3.x is designed to be OS-independent. Accordingly, it does not maintain coupling with DSP/BIOS. This means that DSP/BIOS will use device resources, such as Timer 0 and cache, directly, without making CSL API calls or registering their use with CSL.

The interrupt controller module of CSL 3.x is delivered as a separate library from the other CSL modules. DSP/BIOS users would make use of the interrupt dispatcher in BIOS.

2.2 New Features

2.2.1 Version Check

CSL 3.x implements basic version check APIs. CSL_versionGetID() returns the CSL version ID, and CSL_versionGetStr() returns the CSL version string defined in csl_version.h. These APIs allow the application to check the version of the CSL library being linked in, and compare it to the version of the CSL header files.

2.2.2 Error Reporting

CSL 3.x extends basic support for error reporting. Functional Layer APIs return error codes to the application for handling. Global error codes are defined in csl_error.h.

2.3 Organization

2.3.1 Header and Source Files

Every module in CSL 2.x has two header files: csl_<mod>.h and csl_<mod>hal.h. You must include csl_<mod>.h to use the HAL macros or the modules' functional APIs. The source code is either in a single csl_<mod>.c file or set of <mod>_<funcAPI>.c files.

For example, the header files for the DMA module are csl_dma.h and csl_dmahal.h. The source code could be in csl_dma.c (for C6000), which contains all the functional API definitions in one single file, or it could be in dma_cfg.c, dma_open.c, dma_close.c files (for C5500), which each contain the definition of the individual API. There was no effective separation of implementation code into distinct compile units in CSL 2.x.

In contrast, every module in CSL 3.x has the following header files:

- csl_<mod>.h – Functional layer header file that contains structures, command parameters, and API definitions.
- cslr_<mod>.h – Register layer header file that contains register overlay structure and bitfield definitions. This file is included in csl_<mod>.h.

Functional layer APIs are contained in a set of csl_<mod><FuncAPI>.c files, each containing the definition of an individual API.

For example, the McBSP module would consist of the functional layer header file csl_mcbsp.h, the register layer header file cslr_mcbsp.h, and the C files csl_mcbspOpen.c, csl_mcbspClose.c, csl_mcbspHwControl.c, etc. The C files are provided as a library.

This modular structure makes CSL scalable in terms of memory usage, extensible to different processor architectures, and optimal in terms of memory placement.

2.3.2 Libraries

All modules are built into a single library, with the exception of the INTC module. This is delivered as a separate library in order to avoid conflict with the interrupt dispatcher of an OS. Users who are not using an OS should link in the INTC library in addition to the library for the device in use.

2.3.3 Device Level Modules

There are some device-level modules that provide fundamental services common to the device. These are the CSL, CHIP, DEV, VERSION and INTC modules.

Table 3. Device Level Modules

Field	Description
CSL	System initialization, register layer macro definitions.
CHIP	Chip level field definitions, global function declarations, register read and write macros, global register enumeration.
DEV	Device-level register overlay structure
VERSION	CSL and device revision ID and related APIs.
INTC	Interrupt management and dispatcher.

2.3.3.1 CHIP Module

Register-layer macros and functional APIs related to the chip-specific registers, used for various global configuration settings in the processor, are consolidated into the CHIP module.

The CSL 3.x CHIP module contains two basic register read and write functions, with the format:

```
oldVal = CSL_chipWriteReg(reg , val);
 val = CSL_chipReadReg(reg);
```

For example, to set a value in the ISR register:

```
val = 0x00008000;
 CSL_chipWriteReg(ISR , val);
```

To read the value in the AMR register:

```
val = CSL_chipReadReg(AMR);
```

2.3.3.2 Interrupt Controller Module (INTC)

The interrupt management module in CSL 3.x is named INTC instead of IRQ in CSL 2.x. It was renamed to avoid confusion with the meaning of IRQ with respect to certain devices and cores. This module is covered in more detail in a later section.

Like other modules, the INTC module has an overlay structure defined in the register layer header file cslr_intc.h. This structure consists of the interrupt controller registers, such as event flags, masks and enable registers.

3 Functional Layer Replaces Service Layer

3.1 Data Structures

3.1.1 Module Object and Handle

Most of the peripherals in CSL 2.x are programmed by opening an instance and acquiring a handle to work with that instance. The handle object is defined in CSL itself and a pointer to this object is returned to the user upon successful open. Therefore, CSL internally manages data memory required to work with peripheral instance objects. There is an inconsistency in CSL 2.x, in that some peripherals require acquiring and passing a handle as an argument to the APIs, and others do not.

In CSL 3.x, you must open an instance of the peripheral and acquire the handle before working with it.

Example:

```
hI2c = CSL_i2cOpen(&myI2cObj, CSL_I2C_0, CSL_EXCLUSIVE, NULL, &status);
```

The value of the handle is equal to the address of the instance object, and must be passed to other APIs to work with that instance. If the open request fails, a NULL pointer is returned, along with an error code in the status argument.

An example of an object structure is:

```
typedef struct CSL_I2cObj {
 CSL_I2cRegsOvly regs; /* The register overlay structure of I2C. */
 CSL_InstNum perNum; /* Instance of I2C referred to by the object*/
 } CSL_I2cObj;
```

The regs parameter in the object structure points to the base address of the register overlay structure, which contains all of the registers for the instance.

Unlike CSL 2.x, CSL 3.x does not internally allocate object memory space for all device instances of the given module(s). Instead, the application allocates object memory, either dynamically or at built time, and keeps it active as long as the peripheral is being used.

3.1.2 <Mod> HwSetup Structures Replace <MOD> Config

Every module in CSL 2.x had the <MOD>_Config data structure, which is used by the <MOD>_config() function to program the registers of that module. Also, some modules on certain devices had the <MOD>_Setup data structure, which is used by the <MOD>_setup() function to set up specific parameters of the peripheral.

CSL 3.x combines these data structures into the CSL_<Mod>HwSetup structure. The members of this structure are either the parameters to be configured for the peripheral or one or more pointers to sub-structures that logically group the configurable parameters. This structure is passed to the CSL_<mod>HwSetup() function to configure all the features of the peripheral. The user can add this structure to a debugger watch window to monitor the peripheral parameters.

3.2 Functional Layer APIs

Table 2 gives a quick mapping of CSL 2.x APIs to corresponding CSL 3.x APIs. For a generic representation and explanation of these APIs, refer to the section given.

Table 4. Mapping of CSL 2.x to CSL 3.x APIs

CSL 2.x	CSL 3.x	Comments
CSL_init	CSL_sysInit , CSL_<mod>Init	Refer to Section 3.2.1
<MOD>_open	CSL_<mod>Open	Refer to Section 3.2.2
<MOD>_close	CSL_<mod>Close	Refer to Section 3.2.3
<MOD>_config	CSL_<mod>HwSetup	Refer to Section 3.2.4
<MOD>_getConfig	CSL_<mod>GetHwSetup	Refer to Section 3.2.5
<MOD>_start, <MOD>_stop, <MOD>_pause, <MOD>_resume, <MOD>_setCount, etc.	CSL_<mod>HwControl	Refer to Section 3.2.6
Various query APIs, <MOD>_getCount, <MOD>_xempty, etc.	CSL_<mod>GetHwStatus	Refer to Section 3.2.7
<MOD>_read, <MOD>_write, <MOD>_read32, <MOD>_write32, etc.	CSL_<mod>Read, CSL_<mod>Write	Refer to Section 3.2.8

3.2.1 CSL_sysInit( ) Replaces CSL_init( )

For the users of CSL 2.x, CSL_init() should be the first API call in the application code. This function performs some global initialization tasks (if any) and checks that the CSL library corresponds to the CHIP_<deviceNumber> symbol defined in the application, typically as a compiler option.

Similarly CSL 3.x provides CSL_sysInit(), which may initialize a system data object, or satisfy a device-specific initialization requirement. It should be called once at the beginning of the application.

Apart from this, CSL 3.x provides individual initialization functions for each module, with the function CSL_<mod>Init(). These are called only after CSL_sysInit(), if applicable, and before the use of other functional layer APIs for that module. The initialization APIs are idempotent; calling an initialization API multiple times has the same effect as calling it once.

3.2.2 CSL_<mod>Open( ) Replaces <MOD>_open( )

The generic representation of the CSL 2.x `open' API is:

handle = <MOD>_open (InstanceNum, flags)

- InstanceNum is the peripheral instance number (e.g., TIMER instance 0 or TIMER instance 1, etc.). For peripherals supporting more than one logical channel, a second argument indicates the channel to be opened (e.g., channel number in DMA).
- flags is typically set to <MOD>_OPEN_RESET, instructing CSL that after successful open, reset (power-on reset) all the registers of the peripheral.
- handle is the return value pointing to the instance object
- CSL 2.x internally allocates the memory for the peripheral object instances pointed to by the handle.

In CSL 3.x, the `open' API can be represented generally as:

handle = CSL_<mod>Open(*my<Mod>Obj, my<Mod>Num, myOpenMode, *myHwSetup, *myStatus)

- my<Mod>Obj is the pointer to the module object structure that is defined in user data space and passed by reference
- my<Mod>Num is the peripheral instance
- myOpenMode is used to indicate whether the peripheral should be opened in pin-shared or pin-exclusive mode, which helps arbitrate between peripherals that share pins.
- myHwSetup is the pointer to the CSL_<mod>HwSetup structure defined in user data space to configure the peripheral on successful `open'
- myStatus is the return status

- handle is the pointer to the instance object. A failed open request will return a NULL value.

With CSL 3.x, the memory for the peripheral instance object (i.e., memory for my<Mod>Obj) should be allocated by the user. This API does not initialize any parameters on the peripheral. This is done by CSL_<mod>GetHwSetup(), described later.

3.2.3 CSL_<mod>Close( ) Replaces <MOD>_close( )

The generic representation of CSL 2.x 'close' API is:

<MOD>_close(myModHandle)

- myModHandle is the module handle returned by successful `open' API.

For CSL 3.x, the ' close' API can be represented as:

CSL_<mod>Close(myModHandle)

- myModHandle is the module handle returned by successful `open' API.

The handle is the pointer to the instance object.

3.2.4 CSL_<mod>HwSetup( ) Replaces <MOD>_config( )

As mentioned in Section 3.2.1, configuring peripheral registers in CSL 2.x is achieved by the <MOD>_config() API. The user must define and initialize an instance of the <MOD>_Config structure and then pass its address to this API. A call to this API may look like:

<MOD>_config(myModHandle, &myModCfg)

- myModHandle is the handle returned by the 'open' API
- myModCfg is the initialized Config structure instance

In place of the config function, CSL 3.x provides the CSL_<mod>HwSetup() API for every module. It requires the CSL_<Mod>HwSetup structure as input argument. A call to this API may look like:

myStatus = CSL_<mod>HwSetup(myModHandle, &myHwSetup)

- myModHandle is the module handle returned by successful `open' API
- myHwSetup is the instance of the CSL_<Mod>HwSetup structure
- myStatus is the status or error code returned by the API. A successful setup returns CSL_SOK

3.2.5 CSL_<mod>GetHwSetup( ) Replaces <MOD>_getConfig( )

CSL 2.x provides the <MOD>_getConfig structure to get the peripheral configuration. The format of this API is:

<MOD>_getConfig(myModHandle, &myModConfig)

- myModHandle is the module handle returned by successful 'open' API and
- myModCfg is an instance of the Config structure returned by this API after reading all the registers (which are members of this structure) of the peripheral

CSL 3.x provides CSL_<mod>getHwSetup(), which uses the <Mod>HwSetup structure to get the register parameters. The format of this API is:

myStatus = CSL_<mod>GetHwSetup(myModHandle , &myModHwSetup)

- myModHandle is the module handle returned by successful `open' API
- myHwSetup is the instance of the CSL_<Mod>HwSetup structure
- myStatus is the status or error code. A successful call will return CSL_SOK.

3.2.6 CSL_<mod>HwControl Replaces Multiple CSL 2.x APIs for Peripheral Control

To control the behavior of the peripheral (e.g., start/stop, enable/disable, etc.) or change some frequently used parameters (e.g. source/destination address in DMA or one-shot/auto-reload mode of timer, etc.), CSL 2.x provides individual APIs in many cases. Otherwise, you must pass the entire Config structure with the modified value for such parameters and use the Config API. For complex peripherals, these APIs may be numerous.

In CSL 3.x, the functionality of several APIs is provided by a single API, CSL_<mod>HwControl(), through various commands and input values passed as arguments. This API can be called as:

myStatus = CSL_<mod>HwControl(myModHandle, CSL_<MOD>_CMD_<command name>, <any argument required>)

- myModHandle is the module handle returned by successful `open' API
- CSL_<MOD>_CMD_<command name> is the module specific control command.
- <any argument required> is any module and command-specific argument required to be passed. The function inputs this argument as arbitrary (void *) pointer.
- myStatus is the status or error code. A successful call will return CSL_SOK.

For example, in CSL 2.x:

TIMER_start(hTimer);

Is equivalent to the CSL 3.x:

status = CSL_tmrHwControl(hTimer, CSL_TMR_CMD_START, (void *)&TimeCountMode);

3.2.7 CSL_<mod>GetHwStatus Replaces Multiple CSL 2.x APIs for Parameter Status Query

Many of the modules in CSL 2.x have individual APIs to query for certain frequently used parameters (e.g., MCBSP_xrdy to read the status of the XRDY field). These are generally unique to the peripheral, and could be numerous.

In CSL 3.x the functionality of several APIs intended to get bit or register values is provided by a GetHwStatus API, with various commands to indicate the parameter to be queried.

CSL_<mod>GetHwStatus(myModHandle, CSL_<MOD>_QUERY_<name>, <response>)

- myModHandle is the module handle returned by successful `open' API
- CSL_<MOD>_QUERY_<name> is the module-specific query
- <response> is the return value of the item that is queried

  Example:

  To read the current count value of the TIMER, user may call the following in CSL 2.x:

  currentcnt = TIMER_getCount(myhTim);

  In CSL 3.x:

  CSL_timerGetHwStatus(myhTim, CSL_TIMER_QUERY_COUNT, &currentCnt);

  To read the configuration parameters of DMA, the user may call the following in CSL 2.x:

  DMA_getConfig(myhDma, &readCfg);

  In CSL 3.x:

  CSL_dmaGetHwStatus(myhDma, CSL_DMA_QUERY_CURRENT_HWSETUP, &readHwSetup);

3.2.8 CSL_<mod>Read/Write() Replaces <MOD>_read/write( )

For I/O modules such as McBSP, I2C, GPIO, UART, etc., CSL 2.x provides read/write APIs having the general form:

<MOD>_read(myModHandle, <param1>, <param2>, …)

<MOD>_write(myModHandle, <param1>, <param2>, …)

- myModHandle is the module handle returned by successful `open' API
- <param1>, <param2>, … are the arguments, which vary for every module. For example, GPIO has

PIN ID mask as the second and the last argument, I2C has length, master and slave address, transfer mode, etc. McBSP has only one handle argument but has different APIs to read/write 16-bit and 32-bit words.

In CSL 3.x, all of these modules have a single read and write API, which can be called as:

CSL_<mod>Read(myModHandle, <data_length_command>, <address>)

CSL_<mod>Write(myModHandle, <data_length_command>, <address>)

- myModHandle is the module handle returned by successful `open' API • <data_length_command> specifies the width of the data
- <address> is the memory address of the location to/from which to write/read the data

For example, to read 16-bit data from McBSP, call the following in CSL2.x:

my16BitData = MCBSP_read16(myMcbsp);

In CSL 3.x:

CSL_mcbspRead(myhMcbsp, CSL_MCBSP_WORD_LENGTH_16, &my16BitData);

To write 32-bit data using McBSP, call the following in CSL 2.x:

my32BitData = MCBSP_write32(myhMcbsp);

In CSL 3.x:

CSL_mcbspWrite(myhMcbsp, CSL_MCBSP_WORD_LENGTH_32, &my32BitData);

4 Register Layer Replaces Hardware Abstraction Layer (HAL)

4.1 Common Set of Register Layer Macros Replaces Module-specific HAL Macros

The CSL 2.x HAL layer provides a set of register bit field manipulation macros for each module with the structure <PER>_<MACRO>, where <MACRO> represents ADDR, RGET, RSET, FGET, FSET, etc. (e.g. DMA_RSET(), TIMER_FGET()) Many of the HAL macros are nested up to seven or eight levels, which makes it difficult to understand and debug.

CSL 2.x HAL macros have been replaced by a set of eight register layer field manipulation macros, common to all modules. They are centralized in a single header file, cslr.h. Table 5 describes each of the macros.

Table 5. Register Layer Field Macros

Macro	Description
CSL_FMK(field, val)	Use absolute value `val' to make the field value
CSL_FMKT(field, token)	Use the symbol `token' to make the field value
CSL_FMKR(msb, lsb, val)	Use the absolute value `val' to make the field between `msb' and `lsb' positions
CSL_FEXT(reg, field)	Extract a field value from a register `reg'
CSL_FEXTR(reg, msb, lsb)	Extract a field value from a register `reg' using `msb' and `lsb' positions of the field
CSL_FINS(reg, field, val)	Insert absolute value `val' into the field of the register `reg'
CSL_FINST(reg, field, token)	Use the symbol `token' to insert desired value into the field of the register `reg'
CSL_FINSR(reg, msb, lsb, val)	Insert the absolute value `val' into the field between positions `msb' and `lsb' ; of register `reg'

CSL 2.x also provides <PER>_<REG>_RMK ('Register Make') macros which make up register values via nested FMK ('Field Make') macros. These macros are obsolete in CSL 3.x, however, the same functionality can be achieved by using the register layer macros in conjunction with a pointer to registers in the register overlay structure. This can be done using the format shown in the first example in section 4.

4.2 Register Overlay Structure

CSL 2.x provides HAL macros such as <MOD>_RSETH or <MOD>_FSETH to program a register or a bit field directly. These macros accept the handle acquired through <MOD>_open as an argument.

For example, to program the DMA channel element count register (DMACEN) on C55x, DMA_RSETH (myDmaHandle, DMACEN, Val) macro may be used.

The same can be achieved in CSL 3.x with the help of the handle and register overlay structure. After opening an instance of the peripheral with CSL_<MOD>Open(), the acquired handle has a member called `regs,' which is a pointer to the register overlay structure.

This register overlay structure is defined for every module and is a replica of the module registers in the actual hardware. That is, the sequence and relative offset of these registers in the structure is the same as in the actual memory map of the module registers. By assigning the peripheral base address to the pointer to this structure (which is done by the CSL), all the peripheral registers can be accessed using the structure members. You can directly program the registers individually by using this structure pointer.

Using the same example as above, the DMACEN register may be programmed directly as:

myDmaHandle->regs->CEN = Val;

The register overlay structure members can be used in conjunction with the register layer macros shown in Table 5, to write to registers.

Examples:

To set GO and HLD fields of TIMER control register (TIMER_CTL) to 1 :

```
hTimer->regs->CTL = CSL_FMK(TIMER_CTL_GO, 1) | CSL_FMK(TIMER_CTL_HLD, 1);
```

Check if FRAME field of DMA CSR register is TRUE :

```
if ((CSL_Bool)CSL_FEXT(hDma->regs->CSR, DMA_CSR_FRAME) == CSL_TRUE) ...
```

To set VAL field of WSPR register of watch dog timer (WDTIM) to 0xAAAA

```
CSL_FINS(hWDTim->regs->WSPR, WDTIM_WSPR_VAL, 0xAAAA);
```

To set GO and HLD fields of TIMER CTL register using relevant symbolic constants :

```
hTimer->regs->CTL = CSL_FMKT(TIMER_CTL_GO, START) | CSL_FMKT(TIMER_CTL_HLD, YES);
```

To set EN field of DMA CCR register to START value

```
CSL_FINST(hDma->regs->CCR, DMA_CCR_EN, START);
```

To set prescalar register (PRSC) field in bit positions 0-3 equal to 0x0004 :

```
h->regs->PRSC = CSL_FMKR(3, 0, 0x0004);
```

To return the value in prescalar register (PRSC) field between bit positions 0-3 :

```
return = CSL_FEXTR(h->regs->PRSC, 3, 0);
```

To set 0x0001 value in PRSC register field between bit positions 0-3 :

```
CSL_FINSR(h->regs->PRSC, 3, 0, 0x0001);
```

5 CSL 2.x to 3.x Migration Example

This section illustrates a CSL timer example with CSL 2.x and CSL 3.x APIs.

5.1 Timer Example Using CSL 2.x

```
#include <csl.h>
#include <csl_timer.h>
/* ---------------------------- CSL 2.x Example ------------------------------- */
:
:
TIMER_Handle myhTim  ;   /* Handle is pointer to timer object */
TIMER_Config timCfg = { /* Initialize Config structure */
    TIMER _TCR_RMK (     /* Timer Control Register */
                        TIMER_TCR_IDLEEN_DEFAULT,
                        TIMER_TCR_FUNC_OF(0),
                        :
                        :
                        TIMER_TCR_DATOUT_0),
    0x4000u,          /* Period */
    0x0010            /* Prescalar */
};
TIMER_Config readCfg;   /* Un-initialized, to be used with getConfig API */
:
:
void main()
{
    int newPreScalar;    /* Local variable */
    CSL_init();          /* Initialize CSL */
    /* Open timer0, initialize the registers to power-on reset values */
myTim = TIMER_open (TIMER_DEV0, TIMER_OPEN_RESET);
/* Use initialized structure to configure  the timer */
TIMER_config (myTim, &timCfg);
:
TIMER_start (myhTim);          /* Start the timer */
:
TIMER_stop (myhTim);        /* Stop the timer */
:
TIMER_getConfig (myTim, &readCfg);    /* Read the prescalar value */
newPreScalar = readCfg.prsc * 2;      /* Double the prescalar */
/* Update the register */
TIMER_RSETH (myhTim, PRSC, newPreScalar);
:
TIMER_start (myhTim); /* Start the timer again */
:
:
TIMER_close (myhTim); /* Close the timer, after the usage is over */
}
```

Figure 1. TIMER Example - CSL 2.x

5.2 Timer Example Using CSL 3.x

```
#include <csl_timer.h>
/* ---------------------------- CSL 3.x Example ------------------------------- */
:
:
CSL_Status st = CSL_SOK; /* Status code returned by ‘open’ */
CSL_TimerObj hTimerObj;  /* TIMER object defined in user data space */
CSL_TimerHandle hTimer;  /* Handle will point to TIMER object on
                            successful  ‘open’ */
CSL_TimerHwSetup myTimSetup = { /* Initialize HwSetup structure */
   0xFFFF;             /* Load value – counting starts from here */
   CSL_TIMER_PRESCALE_CLKBY4,    /* Prescalar */
   CSL_TIMER_LOADMODE_RELOAD,    /* Reload mode */
   CSL_TIMER_EMUMODE_RUNFREE,    /* Emulation halt setting */
   CSL_TIMER_EXTCLOCK_ENABLE     /* External clock setting */
};
:
:
void main()
{
   int preScalar;      /* Local variable */
   CSL_sysInit();      /* CSL system initialization */
   CSL_timerInit();    /* Timer module initialization */

   /* Open timer0 */
hTimer = CSL_timerOpen (&hTimerObj, CSL_TIMER_0, CSL_EXCLUSIVE,
                           NULL, &st);
/* Configure timer using initialized HwSetup structure */
CSL_timerHwSetup (hTimer, &myTimSetup);
:
/* Start timer */
CSL_timerHwControl (hTimer, CSL_TIMER_CMD_START, NULL);
:
/* Stop timer */
CSL_timerHwControl (hTimer, CSL_TIMER_CMD_STOP, NULL);
:
/* Read current prescalar value */
CSL_timerHwStatus (hTimer, CSL_TIMER_QUERY_PRESCALE, &preScalar);
preScalar = preScalar * 2;     /* Double the prescalar */
/* Configure timer with new prescalar */
CSL_timerHwControl (hTimer, CSL_TIMER_CMD_SETPRESCALE,
                        (int *) &preScalar);
:
/* Start timer again */
CSL_timerHwControl (hTimer, CSL_TIMER_CMD_START, NULL);
:
:
/* Close the timer after the usage is over */
CSL_timerClose (hTimer);
}
```

Figure 2. TIMER Example - CSL 3.x

6 Interrupt Controller (INTC) Module

This chapter briefly compares the functional APIs provided in the CSL interrupt management module in CSL 2.x and 3.x. Since the IRQ and INTC module implementations are device-specific, depending on the interrupt controller hardware, please refer to the specific CSL API Reference Guide for your product for more details.

Table 6. CSL 2.x IRQ to CSL 3.x INTC API Mapping

CSL 2.x	CSL 3.x	Comments
IRQ_config, IRQ_configArgs, IRQ_getConfig, IRQ_setArg, IRQ_getArg, IRQ_biosPresent	CSL_intcOpen, CSL_intcClose, CSL_intcInit, CSL_intcGetHwStatus, CSL_intcHwSetup	Section 6.1
IRQ_clear, IRQ_test	CSL_intcHwControl	Section 6.2
<MOD>_getEventId	CSL_<mod>GetChipCtxt	Section 6.3
IRQ_enable, IRQ_disable, IRQ_restore	CSL_intcEventEnable, CSL_intcEventDisable, CSL_intcEventRestore	
IRQ_globalEnable, IRQ_gloabalDisable, IRQ_globalRestore	CSL_intcGlobalEnable, CSL_intcGlobalDisable, CSL_intcGlobalRestore	
IRQ_plug/IRQ_hook	CSL_intcHookIsr	Section 6.4
IRQ_map, IRQ_reset	NA	
IRQ_setVecs	NA	
NA	CSL_intcDispatcherInit, CSL_intcPlugEventHandler	Section 6.5

There are some APIs like IRQ_map(), IRQ_reset() in CSL 2.x, which do not have corresponding API in CSL 3.x.

6.1 General Configuration and Query APIs

CSL 2.x IRQ configuration and query APIs, for the most part, are to be used only if DSP-BIOS is present in the application. These APIs are: IRQ_config(), IRQ_configArgs(), IRQ_getConfig(), IRQ_setArg(), IRQ_getArg() and IRQ_biosPresent(). Since CSL 3.x does not have any linkage with DSP-BIOS, it does not support any of these APIs.

There is an equivalent OS-independent set of APIs in the CSL 3.x INTC module. These are the same core set of Functional Layer APIs covered in Section 3.2. A set of register and parameter data structures, similar to other modules, are also employed.

The INTC open API reserves an interrupt event for use, based on the event identifier and vector passed as arguments, and returns a handle to the event.

myHandle = CSL_intcOpen(myIntcObj, myEventId, *myVectId, *myStatus);

- myIntcObj is the pointer to the module object structure that is defined in user data space and passed by reference
- myEventId is the event identifier
- myVectId is the interrupt vector identifier
- myStatus is the return status.
- myHandle is the pointer to the instance object. A failed open request will return a NULL value.

For example, to open an EDMA interrupt at vector ID 4, interrupt 1:

```
vectId = CSL_INTC_VECTID_4;
 hIntcEdma = CSL_intcOpen(&intcObjEdma, CSL_INTC_EVENTID_EDMA3CC_INT1, &vectId , NULL);
```

6.2 APIs for Clearing or Reading Status for an Event

CSL 2.x provides the IRQ_clear() API to clear any pending interrupt that is recorded for a particular event. It provides the IRQ_test() API to read the status of interrupt flag register for a given event.

The similar functionality is provided in the CSL_intcHwControl() API of CSL 3.x. This API also enables setting of other interrupt controller parameters through control commands passed as an argument.

The structure of this API is:

myStatus = CSL_intcHwControl(myHandle, myControlCmd, myCommandArg)

- myHandle is the pointer to the instance object acquired from the open call
- myControlCommand is the command that tells the API what parameter to change
- myCommandArg is the optional argument associated with the command, such as a parameter value
- myStatus is the return status.

Alternatively, the user can achieve the same goal by calling a more specialized lower-level control API. The table shows the control command that would be passed in CSL_intcHwControl to the lower-level API that it would invoke. The event ID is passed as a parameter to the specialized API.

Table 7. CSL 3.x INTC Control Commands and Control APIs

Description	HwControl Control Command	Specialized Control API
Enable Interrupt Event	CSL_INTC_CMD_EVTENABLE	CSL_intcEventEnable
Disable Event	CSL_INTC_CMD_EVTDISABLE	CSL_intcEventDisable
Set Event	CSL_INTC_CMD_EVTSET	CSL_intcEventSet
Clear Event	CSL_INTC_CMD_EVTCLEAR	CSL_intcEventClear
Enable Event Drop Detection	CSL_INTC_CMD_EVTDROPENABLE	CSL_intcInterruptDropEnable
Disable Event Drop Detection	CSL_INTC_CMD_EVTDROPDISABLE	CSL_intcInterruptDropDisable
Invoke User Event Handler	CSL_INTC_CMD_EVTINVOKEFUNCTION	CSL_intcInvokeEventHandle

6.3 APIs to Get Event Ids for a Particular Peripheral

In order to use most of the IRQ APIs in CSL 2.x, the event identifier of the desired peripheral must be passed as an input argument. For devices that map event IDs to specific types of interrupts, these event IDs can be obtained using the <MOD>_getEventId() API for that peripheral.

The corresponding CSL 3.x API is CSL_<mod>GetChipCtxt(), which takes CSL_<MOD>_CHIPCTXTQUERY_EVENTID as the input query-type and returns the event Id requested.

For example, in CSL 2.x:

```
/* Obtain interrupt event id for the opened DMA channel */
 dmaEventId = DMA_getEventId(hDma);
 /* Use eventId as input argument to IRQ APIs */
 IRQ_enable(dmaEventId);
```

In CSL 3.x:

```
/* Obtain interrupt event id for the opened DMA channel */
 CSL_dmaGetChipCtxt(hDma, CSL_DMA_CHIPCTXTQUERY_EVENTID, &myDmaEventId);
 /* Enable the interrupt event for channel opened */
 CSL_intcEventEnable(myDmaEventId, &oldState);
```

6.4 APIs to Hook ISR to CPU Vector Table

CSL 2.x APIs IRQ_plug() (in C5000 CSL) and IRQ_hook() (in C6000 CSL) are provided to hook an ISR directly into the chip interrupt vector table with the necessary code to branch to that ISR. Similar functionality is provided by CSL_intcHookIsr() API in CSL 3.x.

In CSL 2.x, the prerequisite to IRQ_plug()/IRQ_hook() is IRQ_setVecs() API, which sets the base address of the interrupt vector table. The table may be defined in an assembly file in the application and initialized with null/dummy entries, and later filled up by ISR functions using IRQ_plug()/IRQ_hook().

Similarly for CSL 3.x, CSL_chipWriteReg() API should be used to write the interrupt vector table base address into specific device registers.

Example using CSL 2.x:

```
interrupt void myDmaIsr();
 :
 /* Get event ID for opened DMA Channel */
 dmaEventId = DMA_getEventId(hDma);
 :
 IRQ_plug(dmaEventId, &myDmaIsr); /* Hook the ISR to CPU vector table. */
```

Example using CSL 3.x:

```
interrupt void myDmaIsr(); /* Prototype of DMA ISR */
 :
 /* Get event id for opened DMA channel */
 CSL_dmaGetChipCtxt(myhDma, CSL_DMA_CHIPCTXTQUERY_EVENTID, (Uint32 *)&dmaEvtId);
 :
 /* Hook the ISR to CPU vector table */
 CSL_intcHookIsr (dmaEvtId, &myDmaIsr);
```

6.5 CSL 3.x Interrupt Dispatcher

CSL 3.x provides an optional internal interrupt dispatcher, which is an extension of the INTC module that implements the interrupt handler. The dispatcher maintains a table of vectors to be dispatched in response to interrupt assertion. Unlike CSL 2.x, where the IRQ module shares data structures with the DSP/BIOS interrupt dispatcher, this module is OS-independent. However, OS users should the OS-based scheduler when provided.

The steps required to use the internal interrupt dispatcher are listed below:

1. Call CSL_intcDispatcherInit() after calling CSL_intcInit().
2. Call CSL_intcOpen() with the desired interrupt Id as one of the input arguments. This acquires the event handle.
3. Call CSL_intcHwSetup to initialize interrupt parameters (e.g., polarity, type of signal, etc.) for this interrupt event signal. The initialized HwSetup structure object will be passed as an argument.
4. Plug the event handler corresponding to this interrupt using CSL_intcPlugEventHandler().
5. Enable the event by calling CSL_intcEventEnable.
6. Once the interrupt event is no longer needed, release the handle by calling CSL_intcClose().

7 DSP/BIOS Considerations

CSL 2.x has built-in support for DSP/BIOS, in that it reserves an on-chip timer for DSP/BIOS use, and allows DSP/BIOS to manipulate the interrupt dispatch table.

CSL 3.x is OS-independent, so it makes no registration of the use of resources, such as the timer and cache, by DSP/BIOS. DSP/BIOS users should use the interrupt controller/dispatcher that is part of DSP/BIOS, rather than the CSL INTC module. The CSL 3.x INTC module is delivered as a separate library so it can be optionally linked in. DSP/BIOS users should not link in the INTC library, but should use BIOS API calls for interrupt control, including enabling and disabling interrupts.

The DSP/BIOS graphical configuration tool GCONF no longer configures peripherals via CSL API calls, as did earlier versions of the tool.

8 References

1. *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401)
2. *TMS320C55x Chip Support Library API Reference Guide* (SPRU433)
3. *TMS320C54x Chip Support Library API Reference Guide* (SPRU420)
4. *TMS320C672x Chip Support Libraries v3.x API Reference Guide* (SPRC223)
5. *OMAP5912 DSP Chip Support Libraries API Reference Guide* (SPRC199)
6. *OMAP5912 ARM Chip Support Libraries API Reference Guide* (SPRC199)
7. *TMS320C6455 Chip Support Libraries v3.x API Reference Guide* (Pending)

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

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